Project 2 report: Hygienic Subexpression Elimination

by Sal Wolffs s4064542, Hessel Bongers s4368312

Algorithm:
Initially, all lines are read and stored as separate strings. We'll describe the algorithm by looking at what happens to each of those strings. The loop(s) to apply this procedure to all strings given are fairly trivial, and do not affect the actual algorithm.

So, for each string:
First, the string is parsed, recursively, by first building an identifier (by iteratively adding characters from the in stream until a non-alphabetical character is encountered), and then looking at the character which directly followed the identifier to decide how to proceed. If that character was a ',' or ')', the identifier is parsed as a leaf, which forms the base case. If the character was a '(', we enter the recursive case: we call the parser twice, yielding two closed trees (a shared index indicating what has already been parsed, which is properly advanced past '(', ',' and ')', ensures the parser starts at the next identifier which hasn't been parsed yet). Then the identifier is parsed as the root of a tree, which has the two trees yielded by the recursive calls as subtrees.

Side-note: When a tree (node or leaf) is constructed, its hash is calculated and stored in the node (or leaf) itself: for a leaf, this hash is the standard library string hash of the identifier. For a node, it is the hash of its identifier, combined with the hashes of its subtrees (using a short procedure cribbed from the C++ boost library which ensures several desirable properties). This will at a later point allow us to take the hash of each tree in constant time, at the cost of a small increase in space used (but space complexity stays unchanged).

Once the entire expression is parsed, the resulting tree is passed to the CSE algorithm proper, which will use it to build a new tree (for various reasons, this simplifies the algorithm considerably, although of course there might be an equally simple in-place algorithm), which will represent the result of CSE.

The CSE algorithm works as follows: a shared counter and hash table (mapping nodes to their number in the DAG) are set up. Then, the nodes are visited recursively:
First, check if this node is equal to one already visited (and thus represents a second or later appearance of a common subexpression to be eliminated) by looking it up in the hash table. If the hash table already has a number for the node, it is equal to one

already seen. In that case, represent this node in the new tree as a “leaf” with the number associated with it in the hash table as identifier.
If the node does not match any in the hash table, no equal has been seen before. Increment the counter of nodes seen, and add the node being visited to the hash table, with the current value of the counter as associated value. Then, if the node being visited is a leaf, represent it in the new tree as a leaf with the same identifier. If the node being visited is a proper node with children, we hit the recursive case: represent the node in the new tree as a node with the same identifier as the node being visited, with children equal to the CSE representations of the children of the node being visited.

With representation of a parsed node defined as such, the tree resulting from CSE is simply the representation of the root of the tree constructed by parsing the original expression. It then only remains to turn this tree back into a string, which is a rather trivial to_string operation where each node is represented by its identifier and its children, in the proper order with parentheses and commas added as appropriate.

A note about the hash table used during CSE: while conceptually, it maps nodes to their number, in actuality, it only stores references to nodes, to avoid some rather tedious copying work (which would actually bump up the complexity class of the algorithm, in both time and space). Obviously, hashes and equality are computed on the referenced nodes, not on the references themselves.

Complexity Analysis:
Parsing happens in linear time: the shared index ensures that every character is read exactly once, even though the function is recursive, and nothing happens that is not linear in the amount of characters read without recursion: constructing a leaf is linear (copy a string and hash it), and setting up a node is linear (copy identifier, hash) plus constant (set up pointers to trees already accounted for recursively in the complexity analysis) plus constant (combine three hashes).

Since the strings have an average length avg, and the amount of nodes is approximately n/avg, and having longer strings is no worse than having proportionally more nodes, I shall consider string operations to execute in constant time in the discussion below: given an input size n, every doubling in the average length of strings approximately halves the amount of string operations necessary, and the run time comes out the same.

CSE also happens in linear time, but only in the average case: Worst case, it might turn O(n^3), but that only happens if O(n) hashes collide, not pairwise, but all in a single value. Since hashes are designed to collide as little as possible, the average case does not behave at all like the worst case.
More in detail:
First, there's N nodes to be visited (every node in the parse tree once), since the parse tree is traversed and copied with modifications. If a node is not in the hash set yet, this is usually checked in linear time: the hash bucket is often empty.

If the hash bucket is not empty, we need to check the current node against all nodes already in the bucket: first, the full hashes are compared, which takes constant time. If the full hashes are also equal (on 32 bits systems, this is expected to happen a few times, but though I'm not sure of the probability formula, I think it's something ridiculously low, like 2~3 times if we have 100,000 different nodes, which is an upper bound for 200,000 input characters. On 64 bits, there's a less than 1% chance of such a collision for up to 600 million different nodes), we need to compare the underlying trees, which has worst case linear (in the size of the smaller tree) behaviour. However, since this worst case happens extremely rarely (75% of the time, a collision will involve a leaf, which makes the "worst case" complete at once), it's fairly safe to assume this happens "at most once" (though it might in fact happen about twice).
Note that all this starts to fall apart above 200,000 input characters on 32-bit machines, since once birthday collisions start showing up, they show up more and more with increasing N, and specifically, it'll happen a lot more than "at most once".
Long story short: In almost all cases, the hash bucket behaves according to its theoretical average case of constant lookup time. Once or twice per expression, it might need to call an equality check which runs in O(n) rather than O(1), so we neglect this on a per-call basis and allow for a few O(n) equality checks in the final sum of complexities.

Note: the worst-case behaviour becomes visible if we look at what happens if the assumed rarity of collisions turns out false: if by some miracle (or more likely, bug), all hashes come out the same, all nodes end up in the same bucket, and comparing them always takes linear time, so each hash table lookup takes O(n^2) time, and we have to do O(n) lookups. Bad. Luckily, this should never happen if std::hash<string> is any good (which it is).

Moving on: so checking the hash table takes amortized constant time if the node has no match in the hash table. In that case, increasing the counter and adding the node to the hash table also take constant time, and no costly rehashing occurs because we initialized the hash table with a bucket for each node, and a max load factor of 1 (which

will only be reached if the tree is already minimal). So visiting a “new” node takes constant time.

If the node does have a match, this take linear time in the size of the tree of which that node is the root to confirm this: the equality operator has to check every node in the tree. However, none of the children of the node will be visited, since the node will be represented by its DAG tracking number, instead of as a full tree. So we also save a linear amount of visits to nodes. Since we assumed we’d be visiting each of them, we can divide the cost of the equality check over them, which makes the cost constant.

So, we “visit” all N nodes (that is, really visit them, or have them checked as part of an equality test which comes out true), each in constant time. That’s O(n) time needed to build the tree. And O(n) time for that rare equality check that comes up false after a full hash collision (on 32-bit machines). Add the O(n) time from parsing, and the full algorithm runs in O(n).

However, to_string has to traverse the tree, and perform string concatenations on every level. Depending on how well that gets optimized (in place construction or by copying those strings over and over), it could take O(n) or O(n log n). I really hope it’s the former, since otherwise output bumps us up a complexity class, which is a bit of a shame.