A. Our Scheme parser is more bottom-up than top-down. Consider the program

```
(+ (* 3 5) 7)
```

In this case, our Scheme parser receives a list of nine tokens from the tokenizer. By the time the parser has processed seven of the tokens, it has already constructed the subtree corresponding to the expression `(* 3 5)`. In general, our parser joins leaves into subtrees as soon as it can, and it gradually joins these subtrees together to form the final tree. That's what a bottom-up parser does.

[Some students said that the parser processes the tokens left-to-right. That's not a strong answer, because all of our parsers have done that. Some students said that the parser makes a tree; all parsers do. Some students said that a top-down parser starts with a grammar; all parsers do. Some students came up with examples, such as `(+ 3 5)`, that were really too small to illustrate the point.]

B.A. One example is the program

```
(--)
```

The parentheses cause token breaks, but the thing in the middle is not a valid token, according to our grammar. Another example is the program

```
"this string is unfini
```

where the string is not correctly terminated.

B.B. Our parser detects only two kinds of error. One kind is too many open parentheses, as in the example program

```
((+ 3 5)
```

The other kind is too many close parentheses relative to the open parentheses preceding them, as in the example program

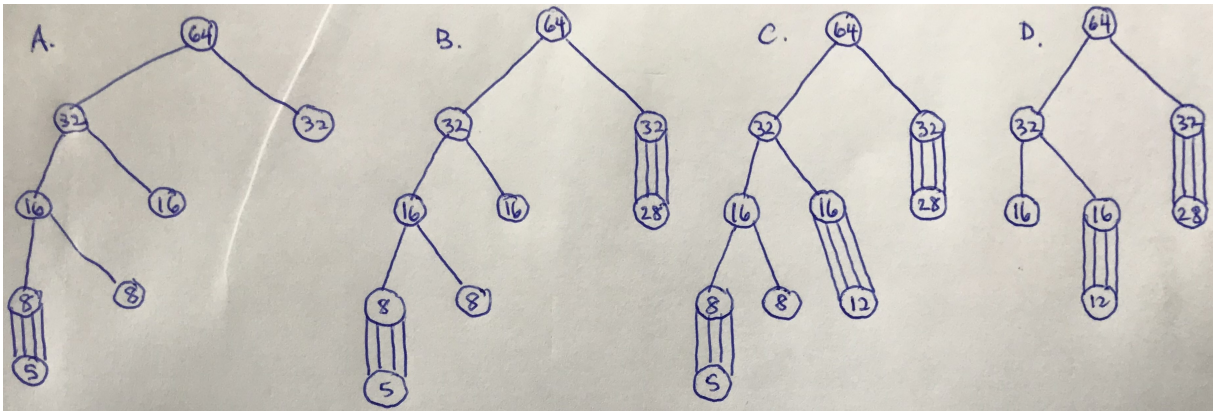
```
(+ 3 5)) (
```

B.C. By the date of this exam, we haven't written the evaluator, but I don't see how we're going to produce a value for the expression

```
(3 5)
```

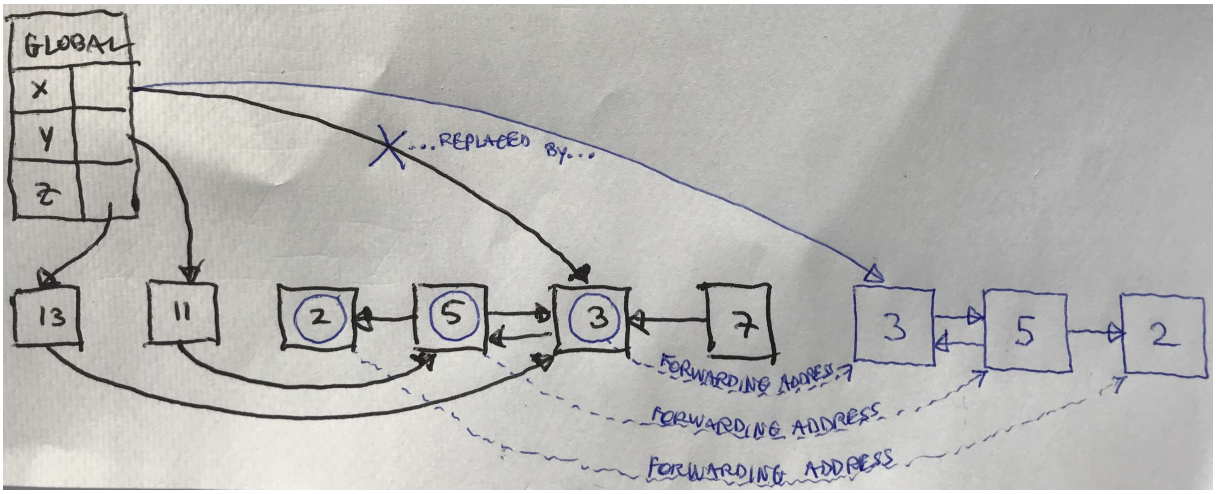
We must eventually check that non-atomic expressions are headed by keywords or functions.

C. The binary trees below show how buddy pairs split, split, split, and coalesce in parts A, B, C, and D of the problem. [We did not discuss the buddy system with enough precision, that there is a unique correct answer to Problem C. Credit was awarded to various answers, as long as they conveyed reasonable splitting and agressive coalescing. My answer wastes a little memory in each allocation, for the sake of simplicity.]



D. In the diagram below, the changes are in blue. The new objects are in the inactive part of memory (which will become active, when the stop-and-copy operation completes). The ordering of 3, then 5, then 2 is important. The forwarding addresses are important.

[Arguably, there should no longer be pointers among the old 3, 5, and 2 nodes, because those nodes have been replaced by a different kind of object (the forwarding address) which might not contain any other pointers. My grading allowed for leaving the pointers or not leaving them.]



E.A. [There are many things one could say, but here are two.] First, static typing detects all type errors at compile time, so that the developer can fix them before deploying the software to users. In contrast, dynamic typing could, if insufficiently tested, show type errors to users. Second, static typing does not require processor time or memory space at run time. In contrast, dynamic typing, by doing its work at run time, increases resource usage by the executing software.

E.B. Dynamic typing can be more flexible than static typing. For example, processing an array of objects, with types that aren't known until run time, is difficult for static typing but easy for dynamic typing.