

In alphabetical order by first name, assign someone to each of the following roles.

Manager: keeps the group on task. Makes sure that everyone gets a chance to participate: if someone hasn't spoken up, asks them what they think. Rotates who speaks first for each question.

Reader: reads each question out loud, when asked by the Manager to do so. The reader can choose to skip over long-winded details, but should definitely read all of the main portions of the questions out loud.

Recorder: writes in the answers that the group agrees on for each question.

Presenter: reports answers back to the main group when we come back together. Each group won't report back every question; I'll pick one or two groups to report for each question, and the Presenter will speak for the group.

(If 3 people, reader/presenter are same person.)

Fill in your names here:

- Manager: _____
- Reader: _____
- Recorder: _____
- Presenter: _____

BNF example:

```
<sentence> ::= <subject> <predicate>
<subject>  ::= <article> <noun>
<predicate> ::= <verb> <article> <noun>
<verb>     ::= ran | ate
<article>  ::= the
<noun>     ::= elf | hobbit | cake
```

Some definitions:

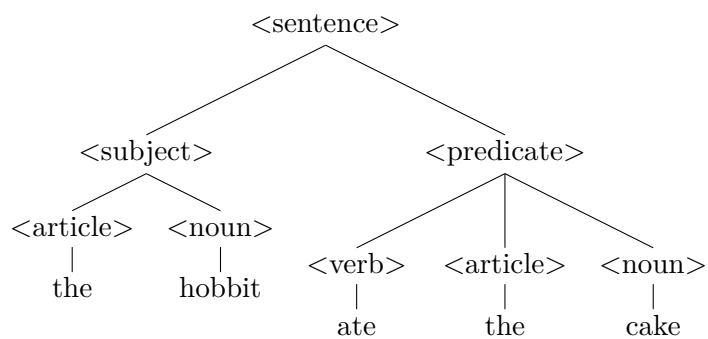
1. Terminals: symbols that actually appear in the language: (, a, b, c, 3, . etc.
2. Non-terminal: symbols that are used to represent symbols or collection of symbols in the language: <subject>, <verb>, etc.
3. Start symbol: One of the nonterminals; in this case, designate as: <sentence>

Sample derivation: apply a rule at a time, as you like:

```
<sentence> => <subject> <predicate>      First rule
            => <article> <noun> <predicate> Second rule
            => the <noun> <predicate>      Fifth rule
            ... => the hobbit ate the cake
```

1. BNF notation isn't well standardized. That said, there's a difference in how I'm using ::= and ==>. What's the difference?

Here is a parse tree to match the derivation above:



Here is another grammar:

Grammar: $\langle B \rangle ::= 0\langle B \rangle \mid 1\langle B \rangle \mid 0 \mid 1$

2. Now do your own derivation: derive the string 010.

3. Now draw a parse tree based on your derivation.

Here is another grammar:

```
<S> ::= <S> + <S> | <S> * <S> | number
```

4. Provide two derivations for the string $2 * 3 + 4 * 5$, each of which results in a different parse tree than the other.

5. An interpreter traverses the parse tree in order to execute the program, and a compiler traverses the parse tree in order to generate code. Suppose we use an interpreter strategy to assign a value to every node in the parse tree as follows:
1. If a node is a number, its value is that number.
 2. If a node is an operation (+ or *), it does not have a value assigned to it directly.
 3. If a node is an $\langle S \rangle$, look at its three children. Its middle child must be an operation. The value for that node is the value obtained by applying the operation to the left child and right child values.

(If the above seems confusing, it is merely intended to describe precisely what is likely already your intuitive understanding of how the tree is supposed to work.)

For each of your parse trees in the previous exercise, determine what the value assigned to the root node should be.

6. A grammar that can result in two different parse trees for some string is referred to as *ambiguous*. From a PL perspective, why is an ambiguous grammar problematic?

7. Here are two different grammars, each of which are variations on representing arithmetic operations. Which one would result in parse trees that better capture our usual order of operations? Why? (*i* is used to abbreviate representing an integer.)

Grammar 1:

```
<E> ::= <E> + <T> | <T>
<T> ::= <T> * <P> | <P>
<P> ::= i | ( <E> )
```

Grammar 2:

```
<E> ::= <E> + <T> | <E> * <T> | <T>
<T> ::= i | ( <E> )
```

Your answer:

Parsing Scheme

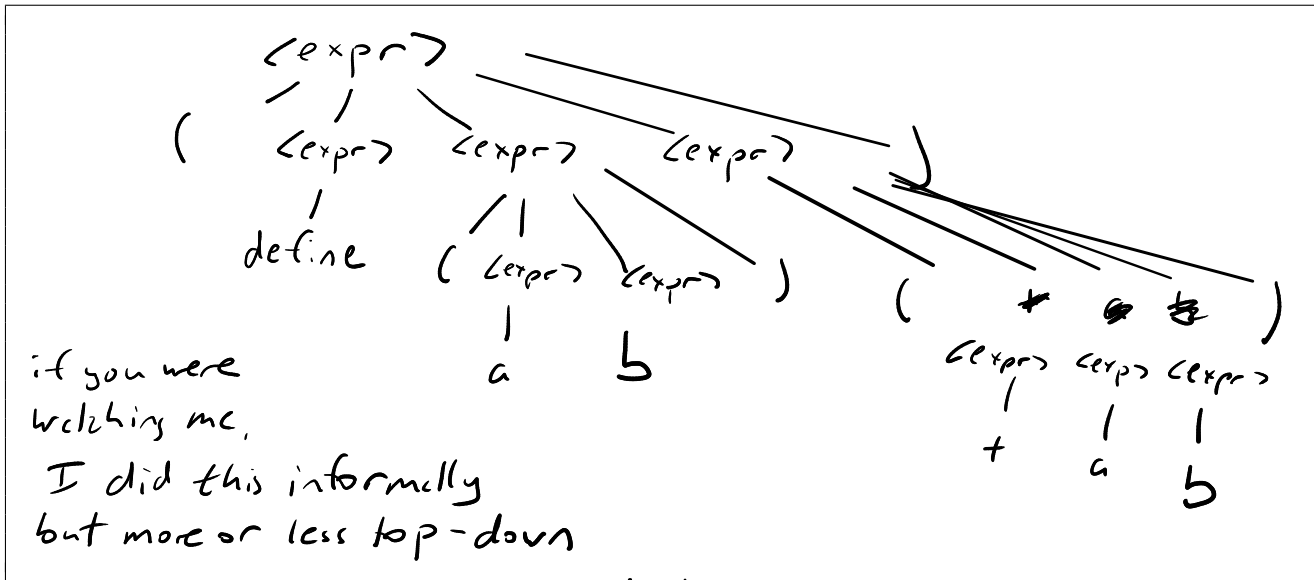
Here is an abbreviated (and possibly flawed) grammar for Scheme:

<expr> ::= atom | (<expr>*)

where the asterisk above means 0 or more repetitions.

8. Write down both a derivation and the matching parse tree for the following Scheme program.

```
(define sum
  (lambda (a b)
    (+ a b)))
```



Alg on next page is essentially bottom up.

9. Suppose you needed to instruct a computer to do what you just did. (You'll eventually need to do exactly that.) Describe at a high level how you were able to come up with the parse tree for the previous problem.

Your answer to the previous question defined a *parsing algorithm*. There are many well-known parsing algorithms that are used in practice, with a variety of pros and cons associated with them. Here is a parsing algorithm that isn't very generalizable, but works wonderfully with Scheme:

Initialize a stack

while not end-of-file **do**

 Read next token

if token is not a right paren **then**

 push token onto stack

else

 Initialize a new subtree with root <expr>

repeat

 pop token from stack

 add popped token as leftmost child to subtree

until stack.peek() is a left paren or stack is empty

if stack is empty **then**

 throw a parsing error and quit

else

 pop off the left paren, throw it away

 push the new subtree onto the stack

end if

end if

end while

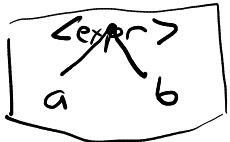
Verify that there is no subtree in progress; throw parsing error if so

10. Parse the sample Scheme program again, but this time do so following the algorithm above. Show your stack as you go. Use the next page for space if you need it.

```
(define sum
  (lambda (a b)
    (+ a b)))
```

→

stack: (define sum (lambda a b



11. This parsing algorithm is Scheme-specific, so it doesn't really have a universal name. Make up something appropriate and write it below.

