Constructing a Parse Tree

Initial Conditions

Before we begin we need to have two things:

1. A Grammar
2. A string we wish to parse

So then what is the procedure:

1. Use the grammar to produce a derivation resulting in the given string
2. Use the derivation to produce the parse tree

Example 1:

We have the following grammar:

```
<S> ::= <round> <square> | <outer>
<round> ::= ( <round> ) | ( )
<square> ::= [ <square> ] | [ ]
<outer> ::= ( <outer> ] | ( <inner> ]
<inner> ::= ) <inner> [ | ) [
```

We will derive the following string: (())[[]]

Example 1: Producing the derivation

A derivation is produce using the following steps:

1. Start with the start rule and select one of its options
2. We then continue to replace each rule until we reach a terminal working from left to right.
3. We repeat step 2 until all non-terminals are replaced by terminals, and we have produced the target string.

Goal: Derive `( ( ) ) [ [ ] ]`

```
<S> => <round> <square>
    => ( <round> ) <square>
    => ( ( ) ) <square>
    => ( ( ) ) [ <square> ]
    => ( ( ) ) [ [ ] ] --> Finished
```

Example 1: Producing the parse tree

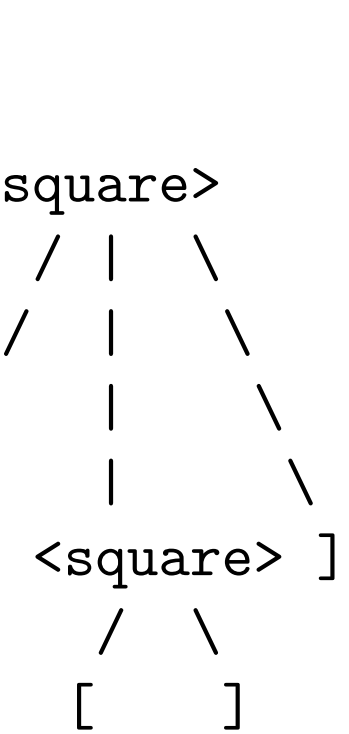
<S> => <round> <square>

=> (<round>) <square>

=> (( )) <square>

=> (( )) [<square>]

=> (( )) [[ ]]



Example 2

Grammar:

```
<S1> ::= <S1> + <S2> | <S2>
<S2> ::= <S2> * <S3> | <S3>
<S3> ::= ( <S1> ) | a | b | c
```

String: `a + b * c`

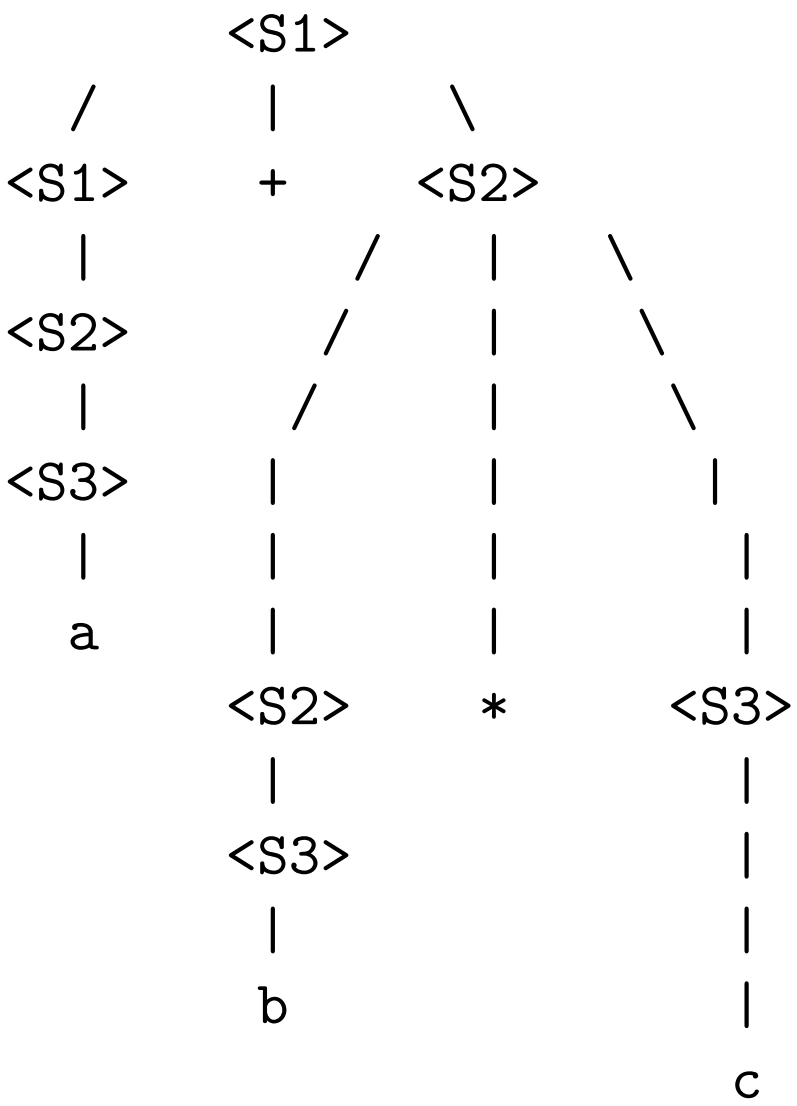
Example 2: Derivation

Goal: Derive `a + b * c`

```
<S1> => <S1> + <S2>
     => <S2> + <S2>
     => <S3> + <S2>
     => a + <S2>
     => a + <S2> * <S3>
     => a + <S3> * <S3>
     => a + b * <S3>
     => a + b * c
```

Example 2: Parse tree

```
<S1> => <S1> + <S2>
     => <S2> + <S2>
     => <S3> + <S2>
     => a + <S2>
     => a + <S2> * <S3>
     => a + <S3> * <S3>
     => a + b * <S3>
     => a + b * c
```



We can also derive a string from the bottom up

Goal: Use Example 2 grammar to derive `a + b * c`

In This case we start with the rightmost terminal and continue to replace with non-terminals until we reach the start rule.

```
      => a + b * c
      => a + b * <S3>
      => a + <S3> * <S3>
      => a + <S2> * <S3>
      => a + <S2>
      => <S3> + <S2>
      => <S2> + <S2>
      => <S1> + <S2>
<S1> => <S1> + <S2>
```

We then build the parse tree starting from the bottom

Constructing Abstract Syntax Trees

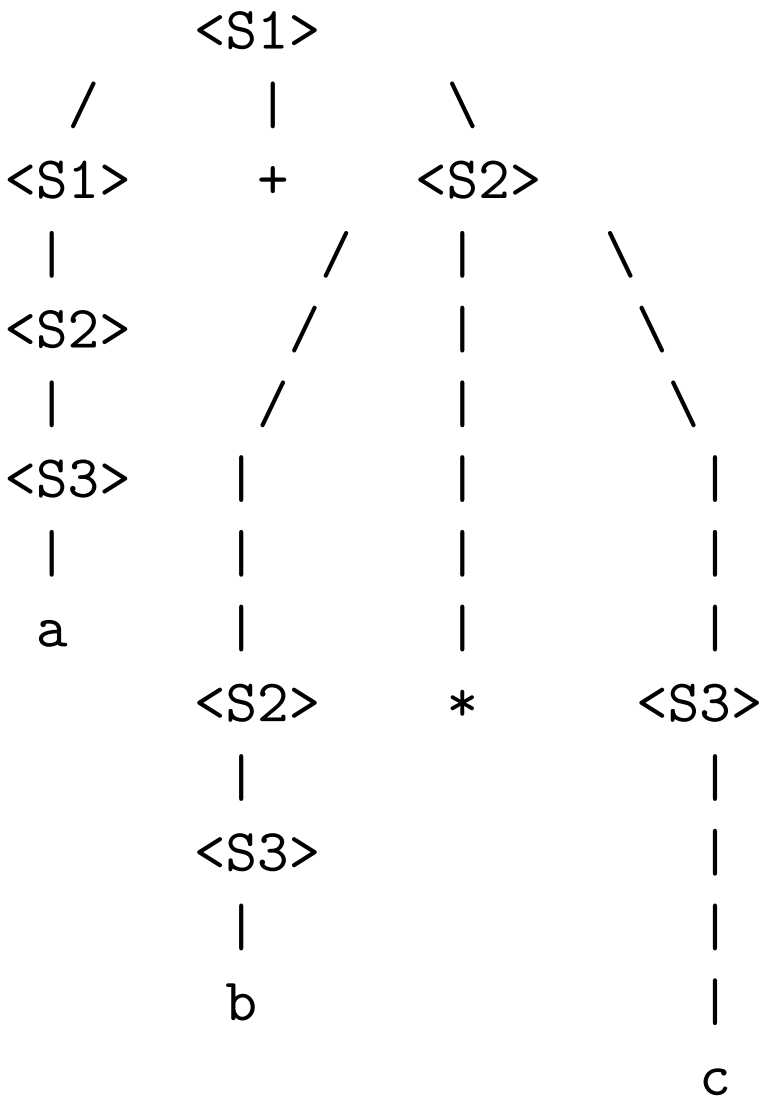
Parse Trees to ASTs

An Abstract Syntax Tree (AST) is a simplified form of a Parse Tree which is useful for interpreting/converting code from one language to another. Thus, it is useful for the compiling process.

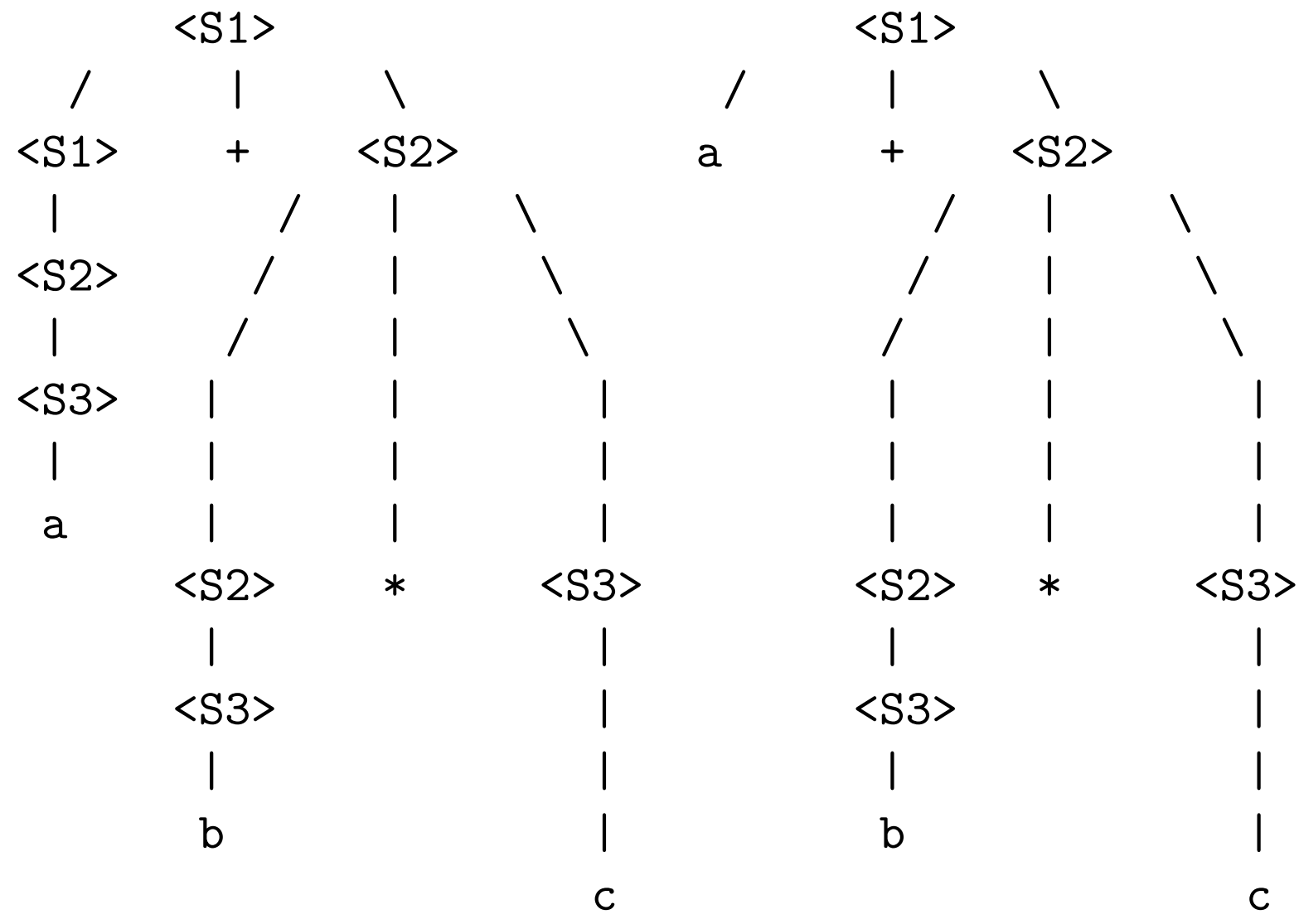
An AST typically is a Binary Tree and requires that there are no non-terminal symbols left in the tree, and this then requires that we have the following:

- A Parse Tree
- A notion of traversing the tree (we will assume an In Order traversal)

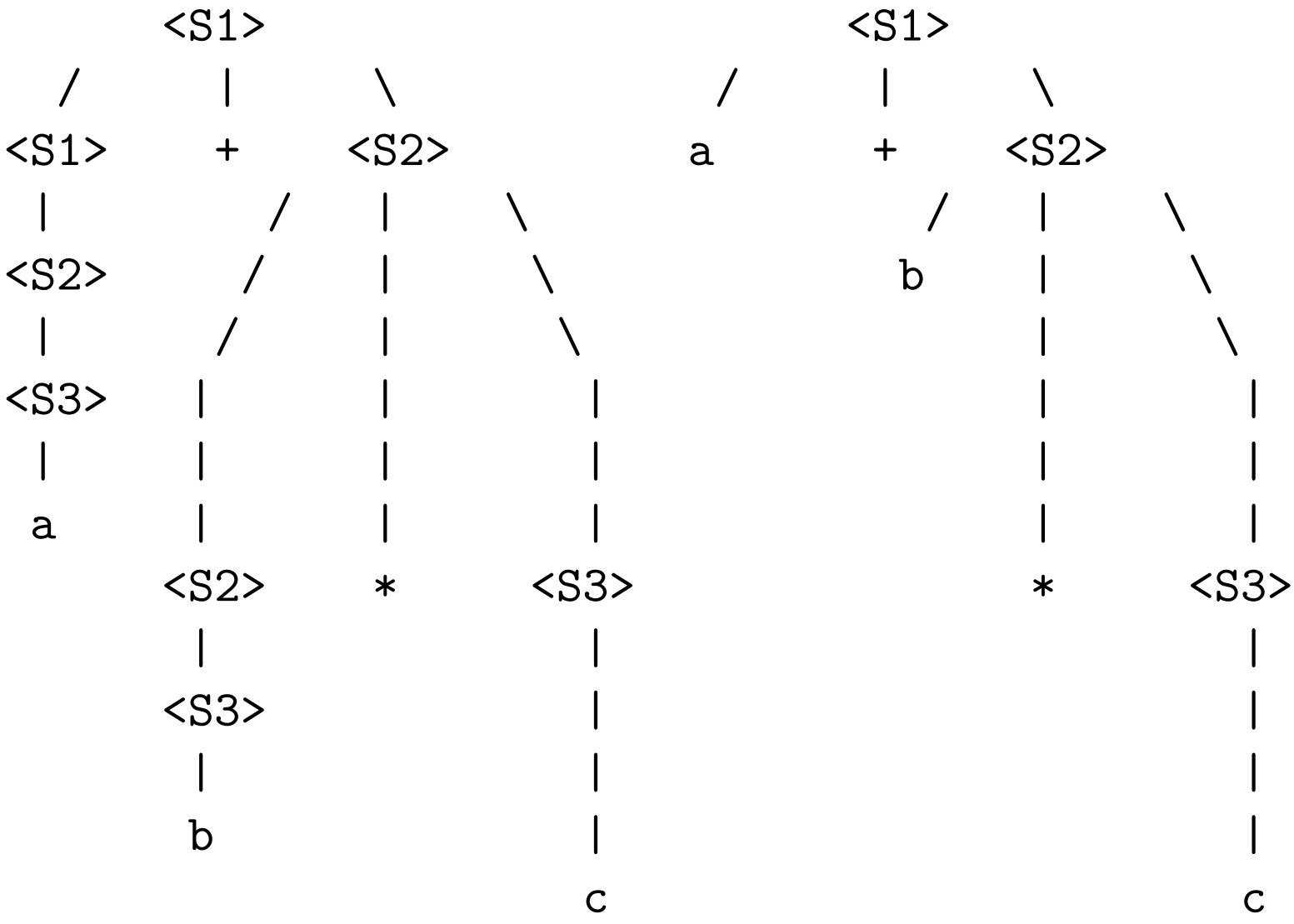
The Conversion Process



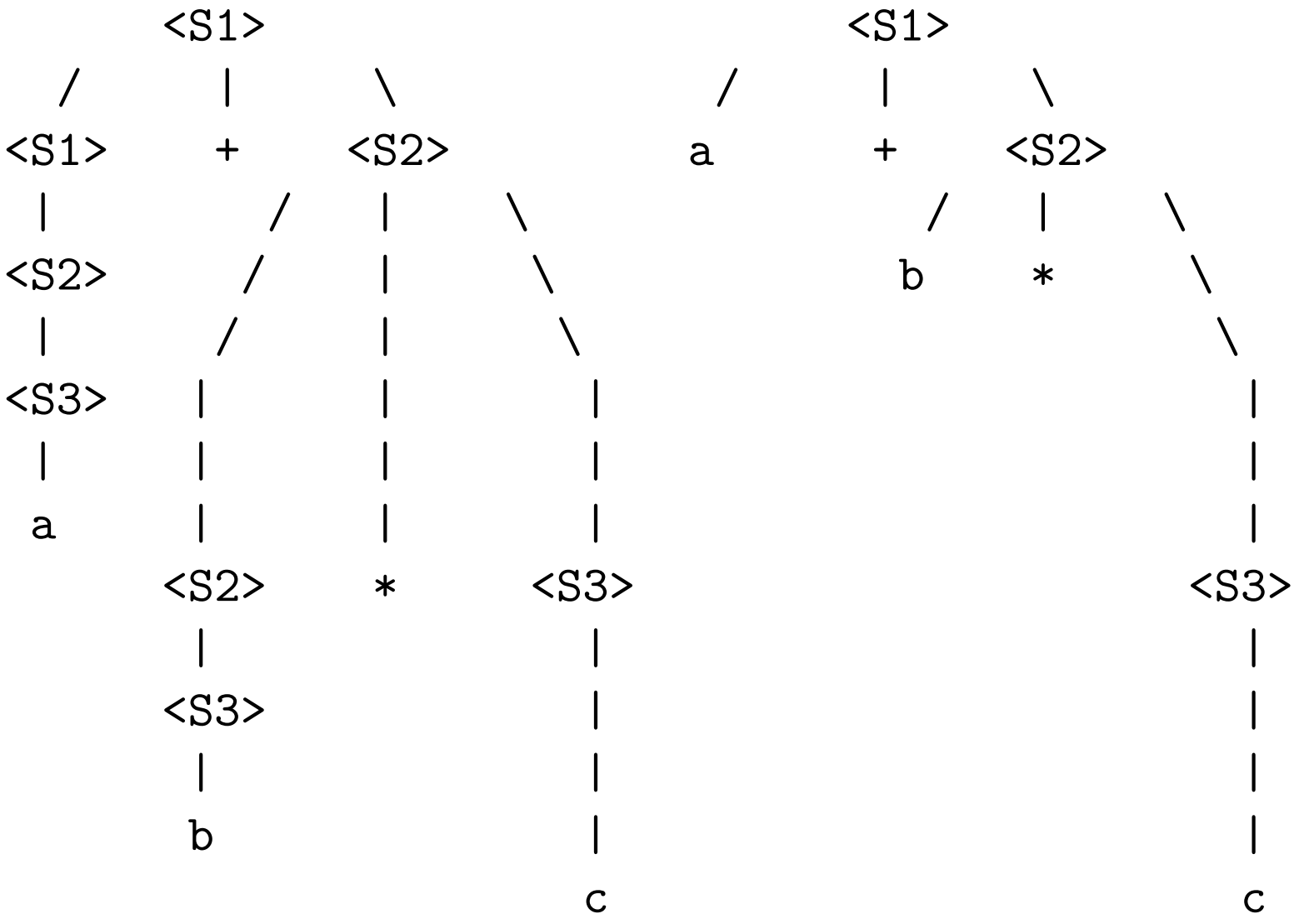
Example



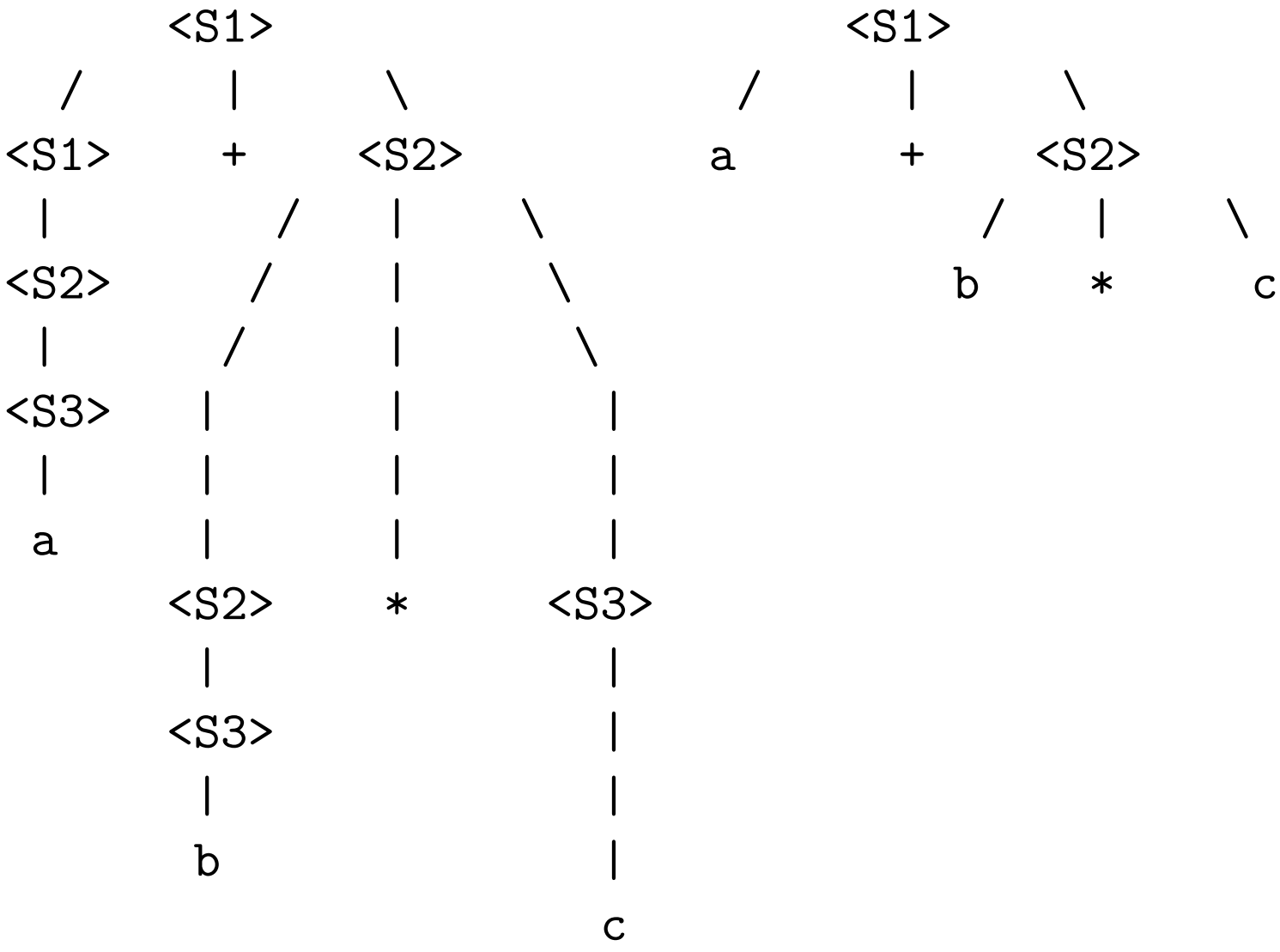
Example



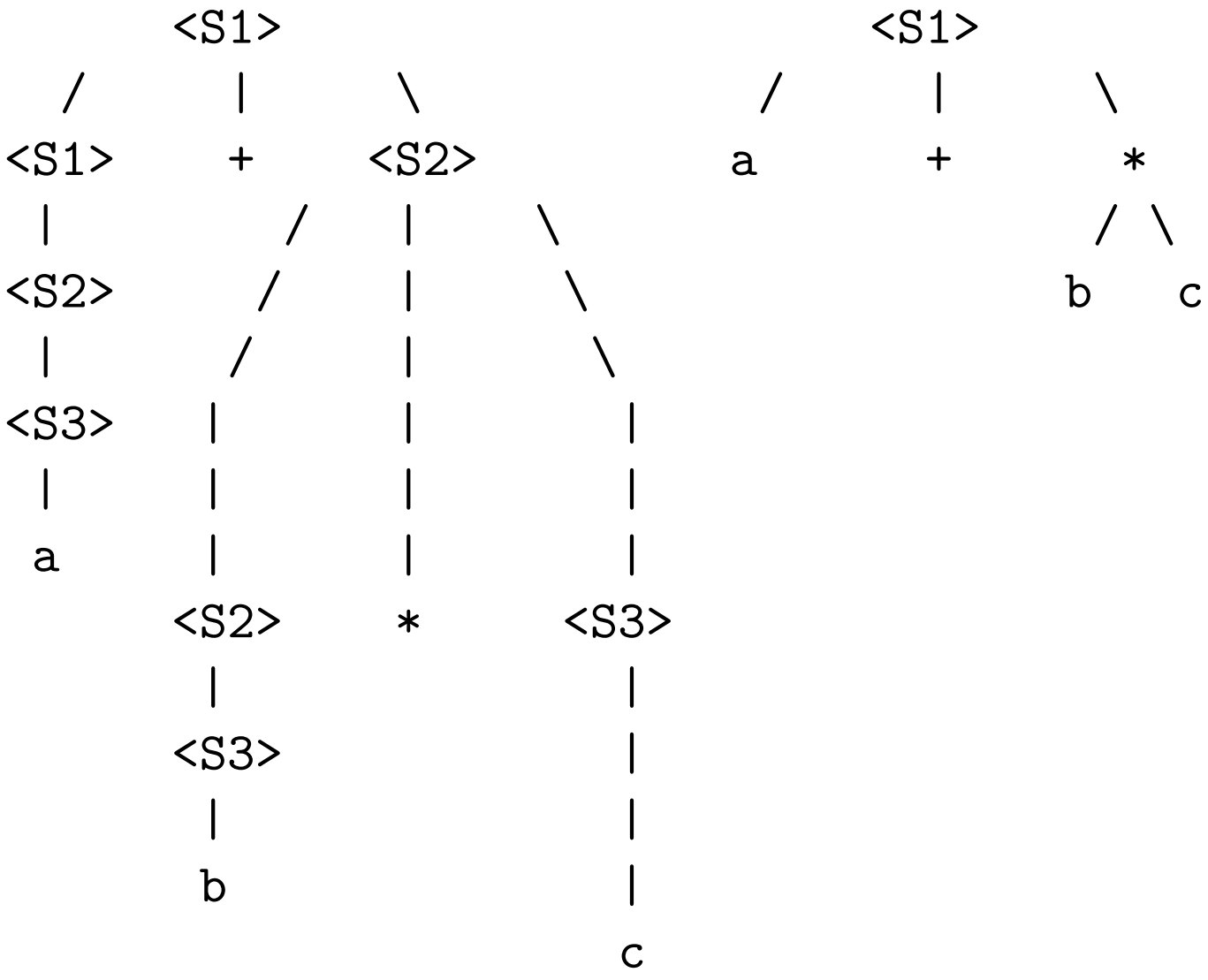
Example



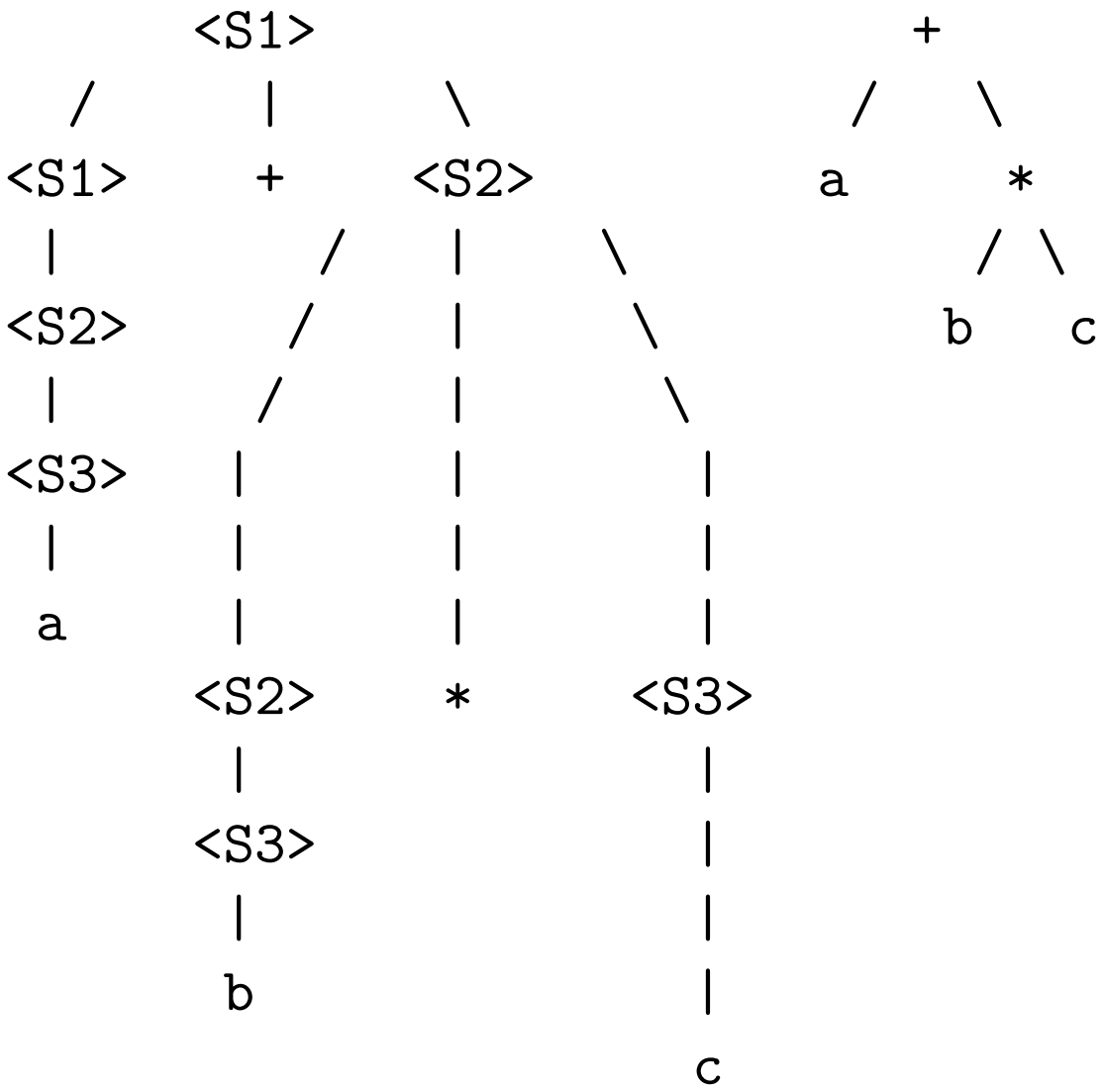
Example



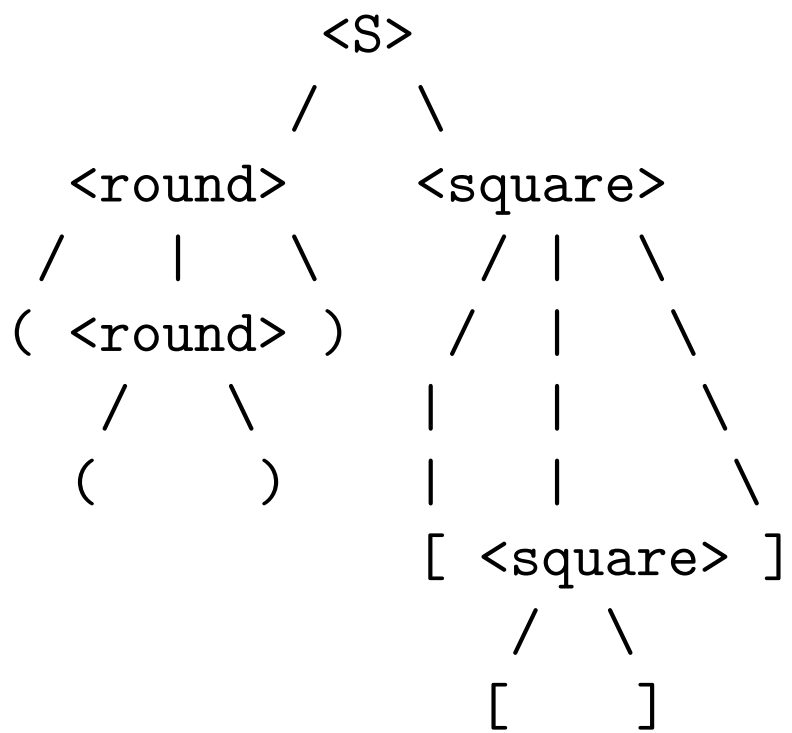
Example



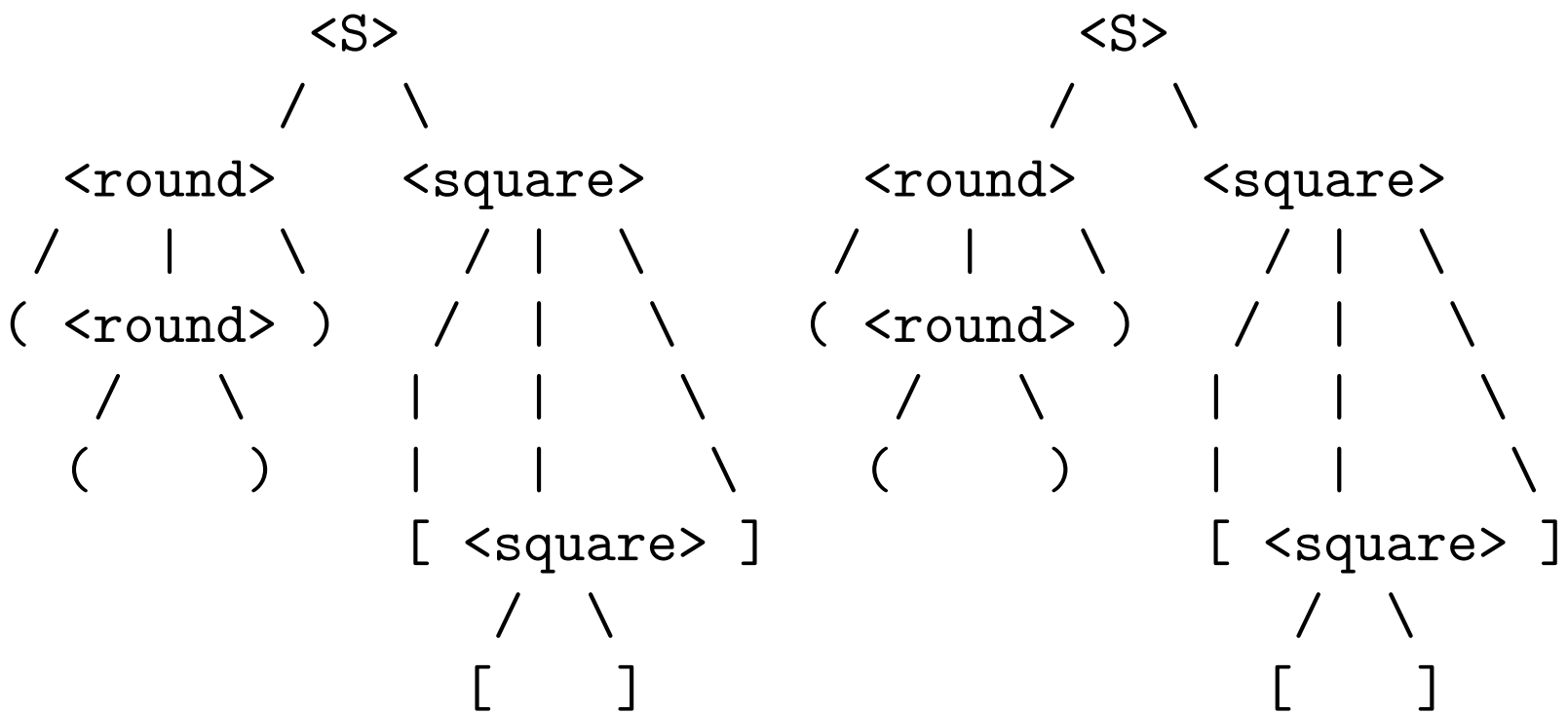
Example



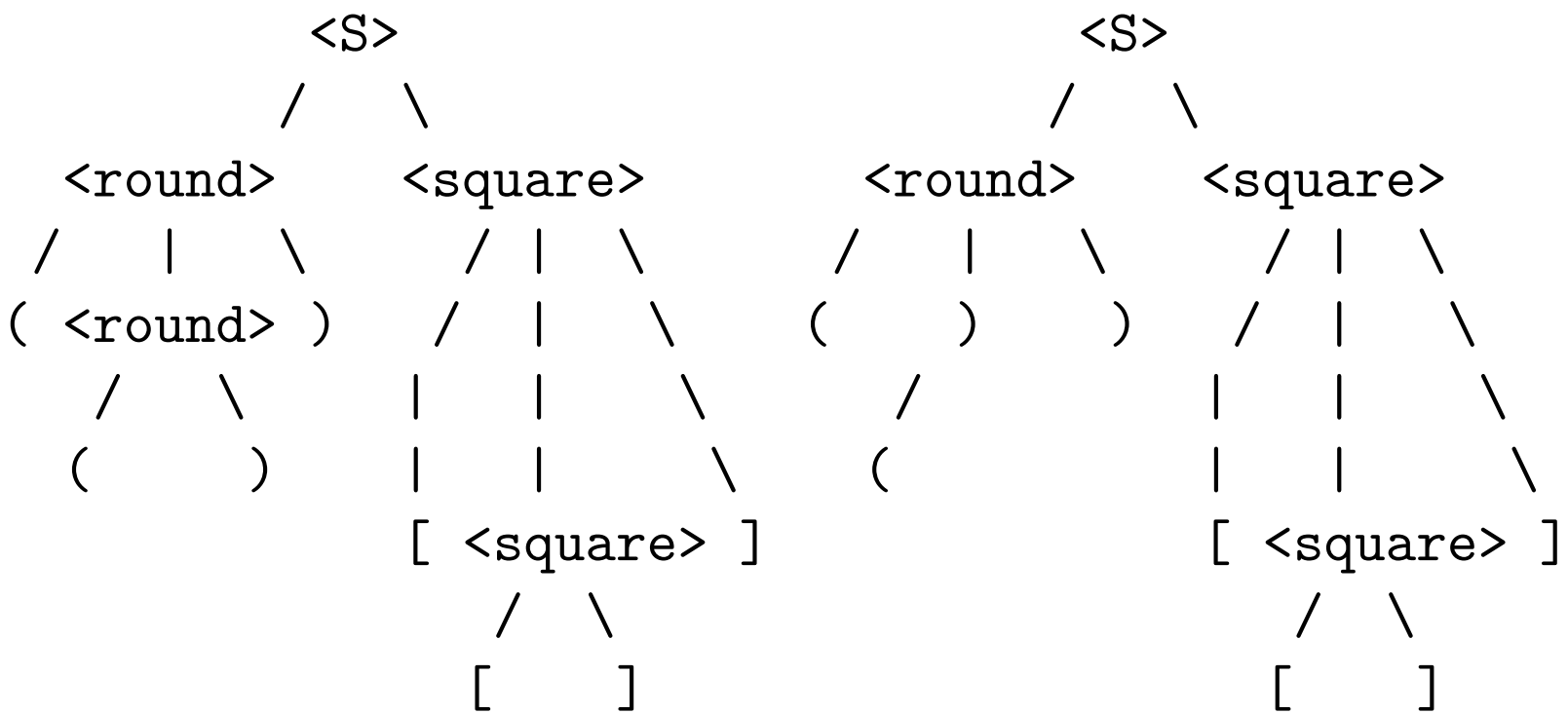
Example



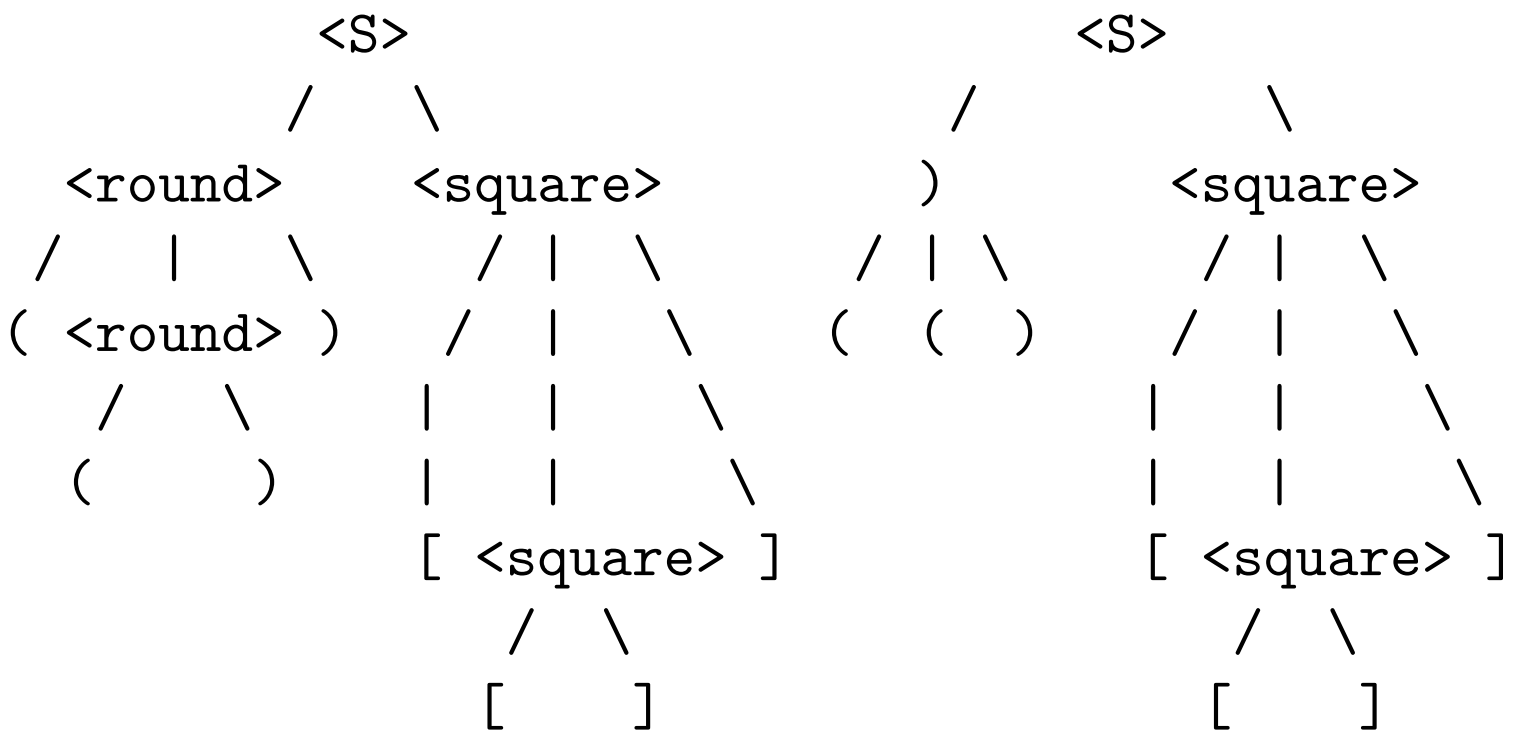
Example



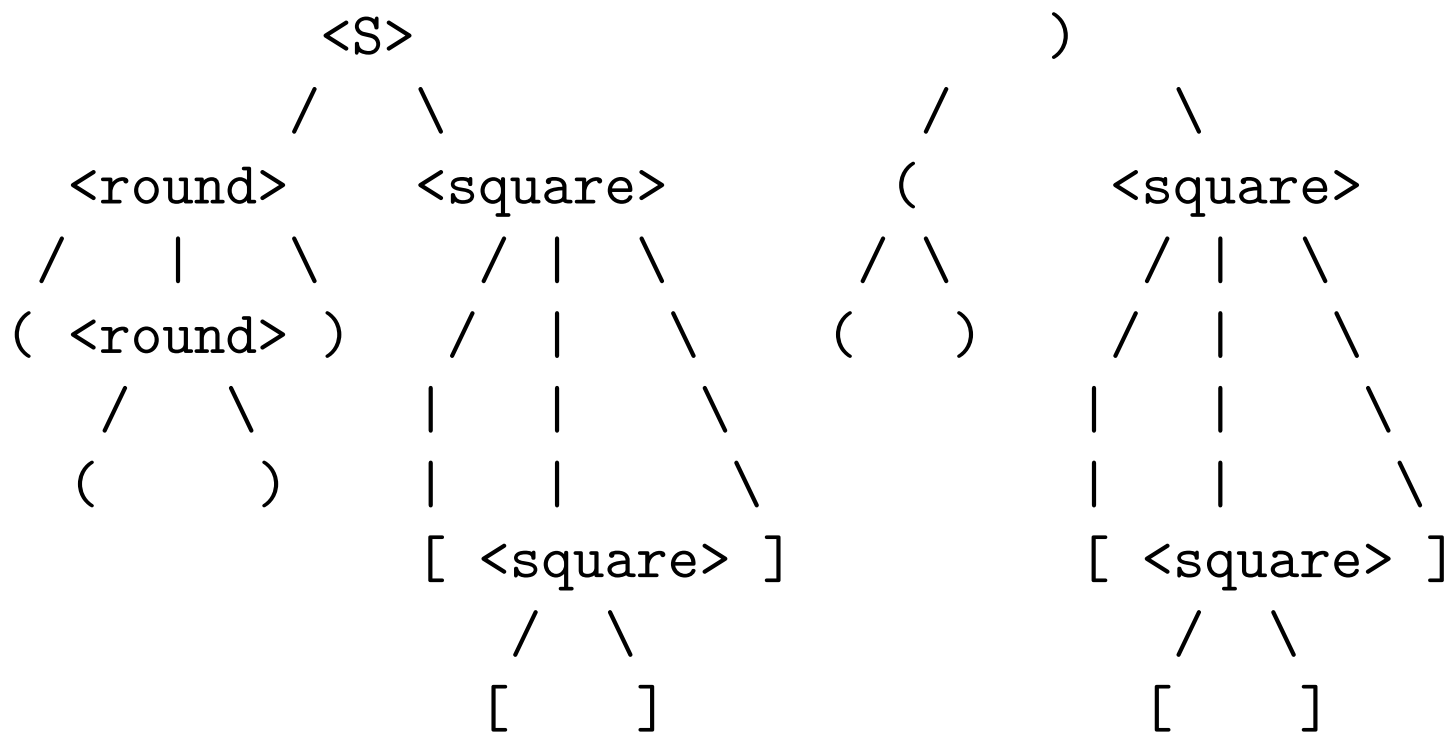
Example



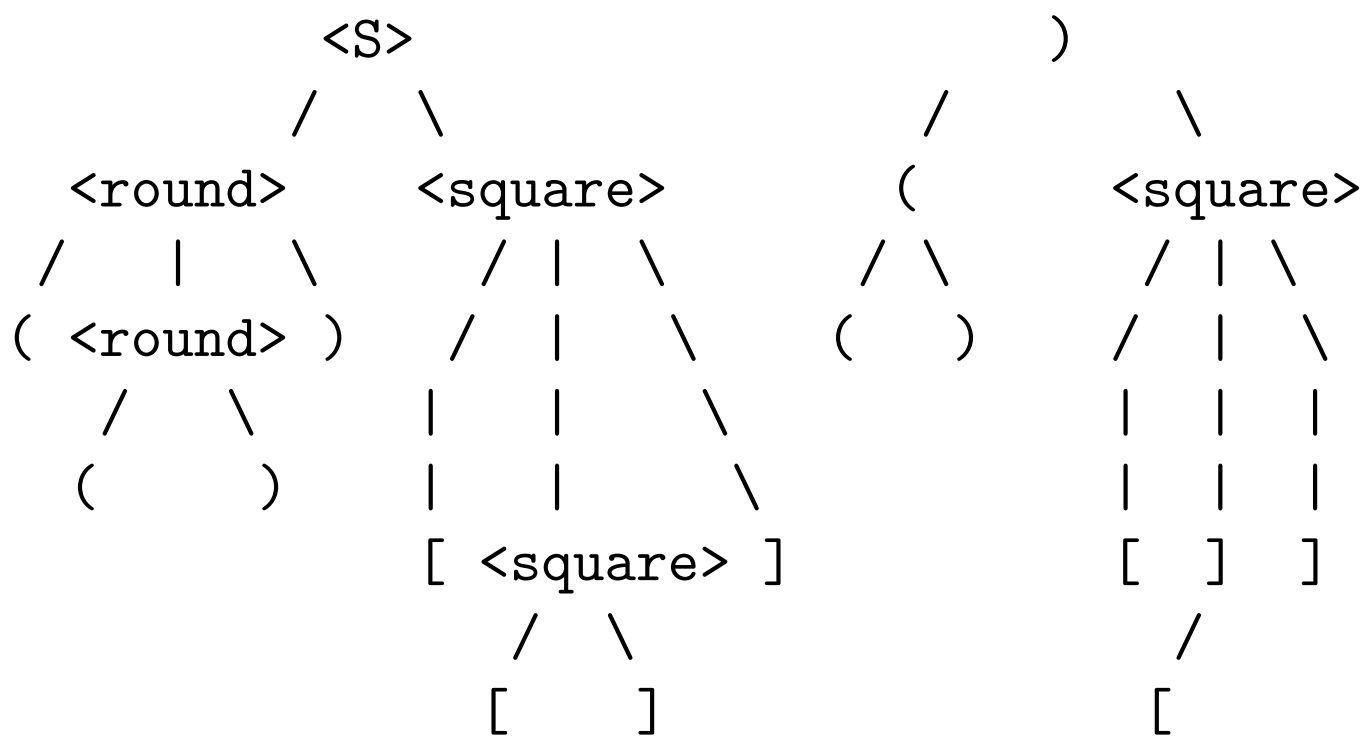
Example



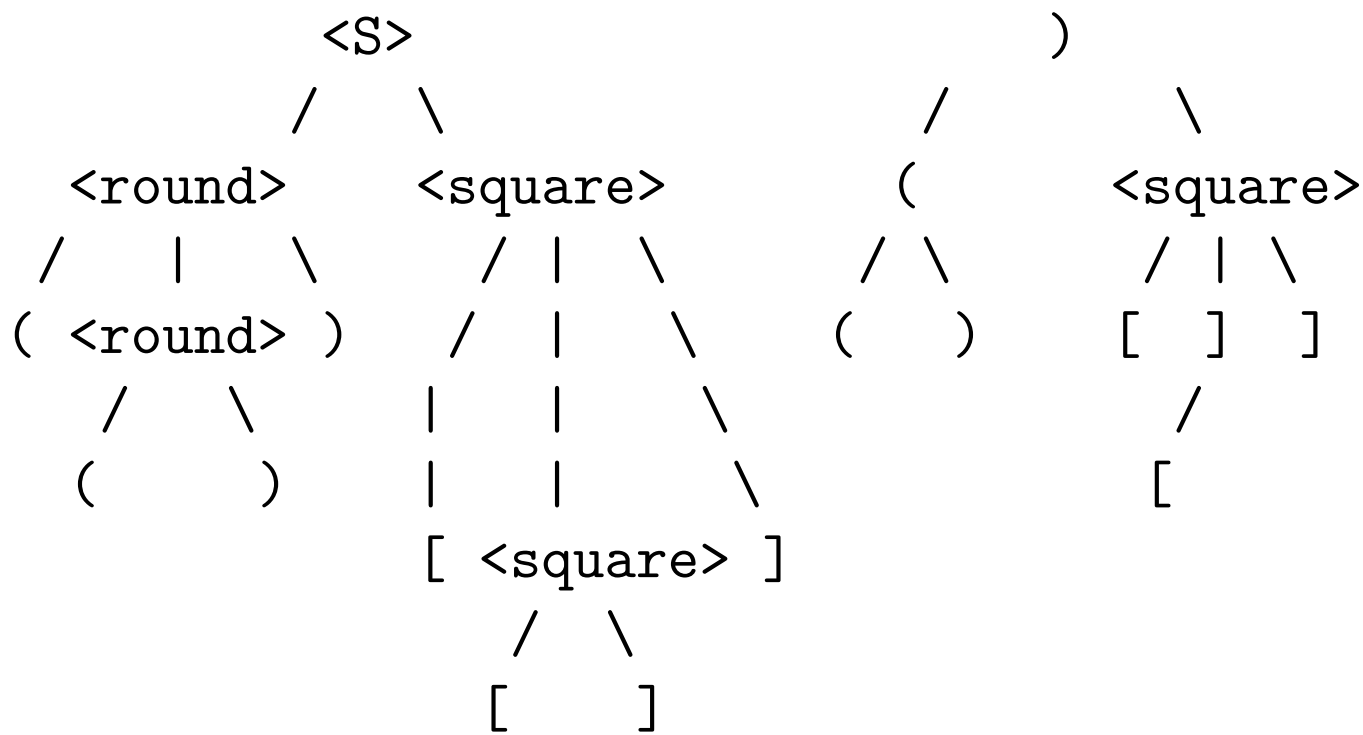
Example



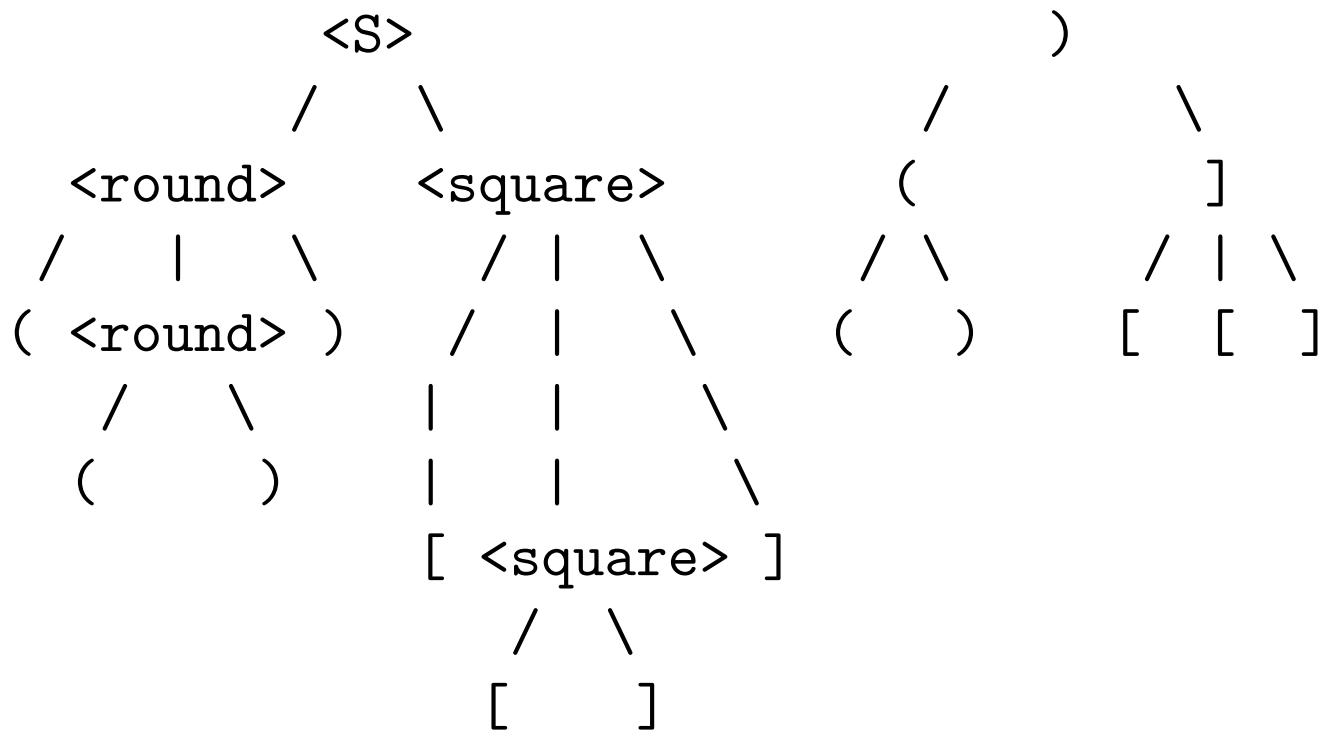
Example



Example



Example



Example