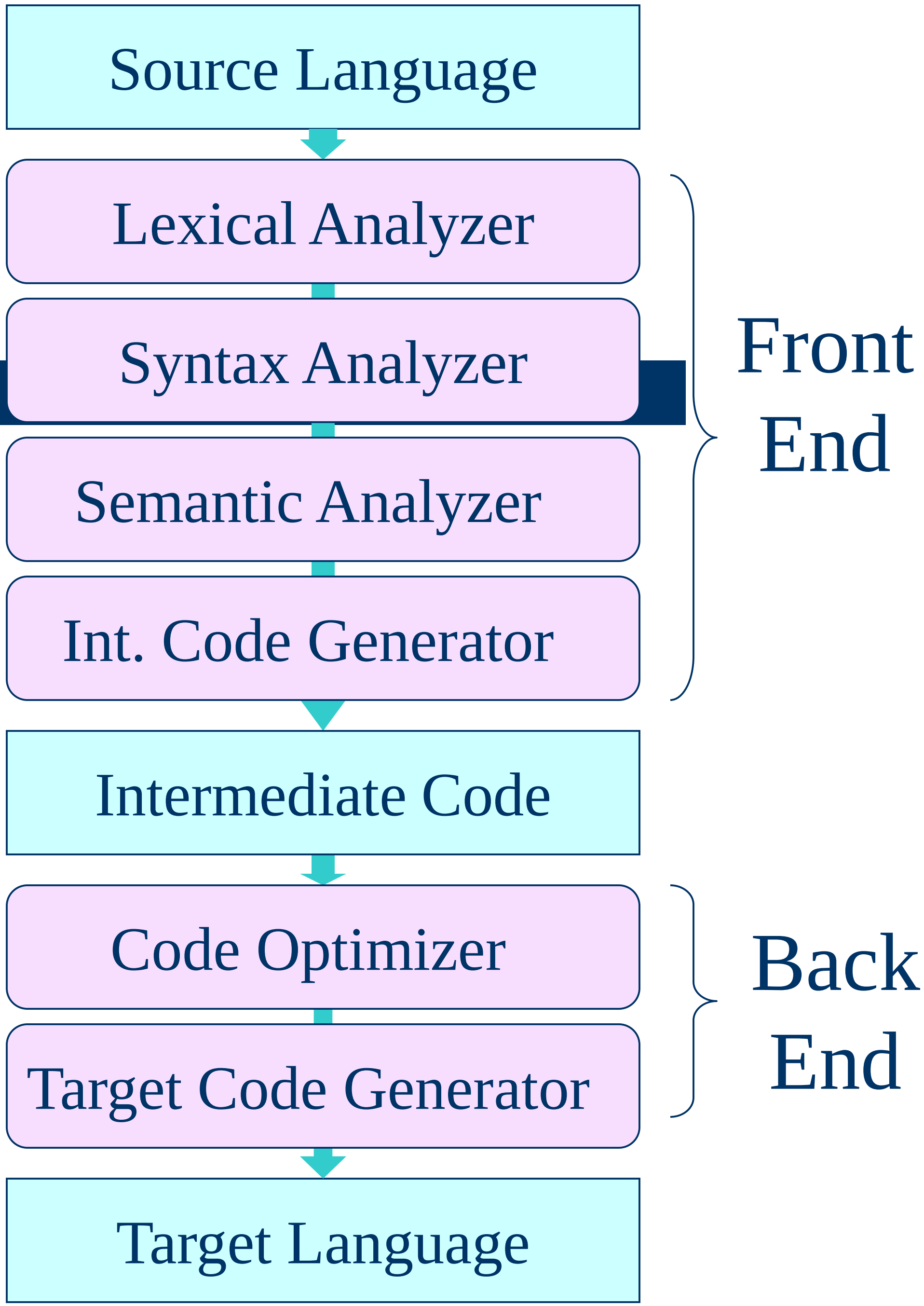
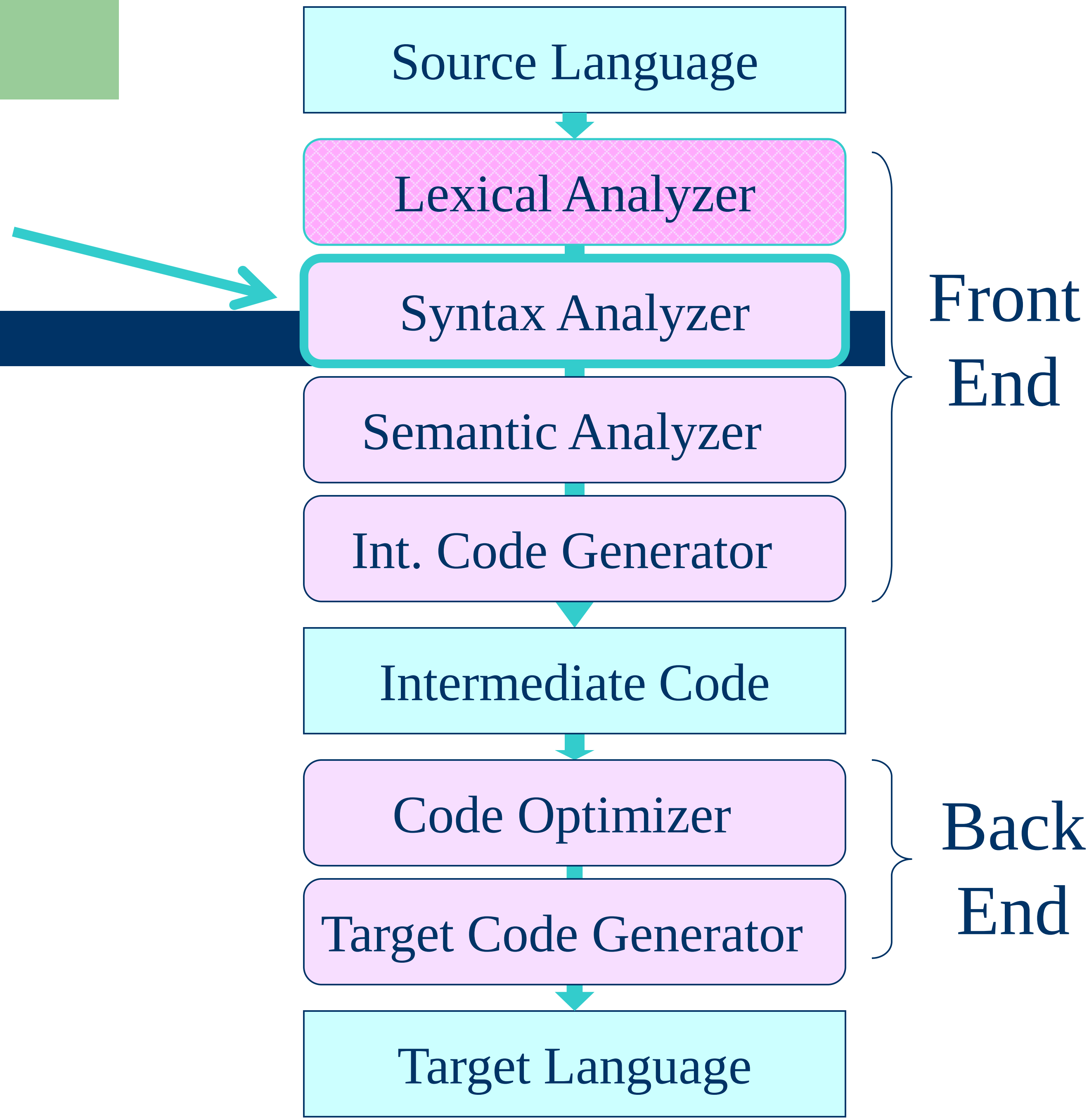
Structure of a Compiler

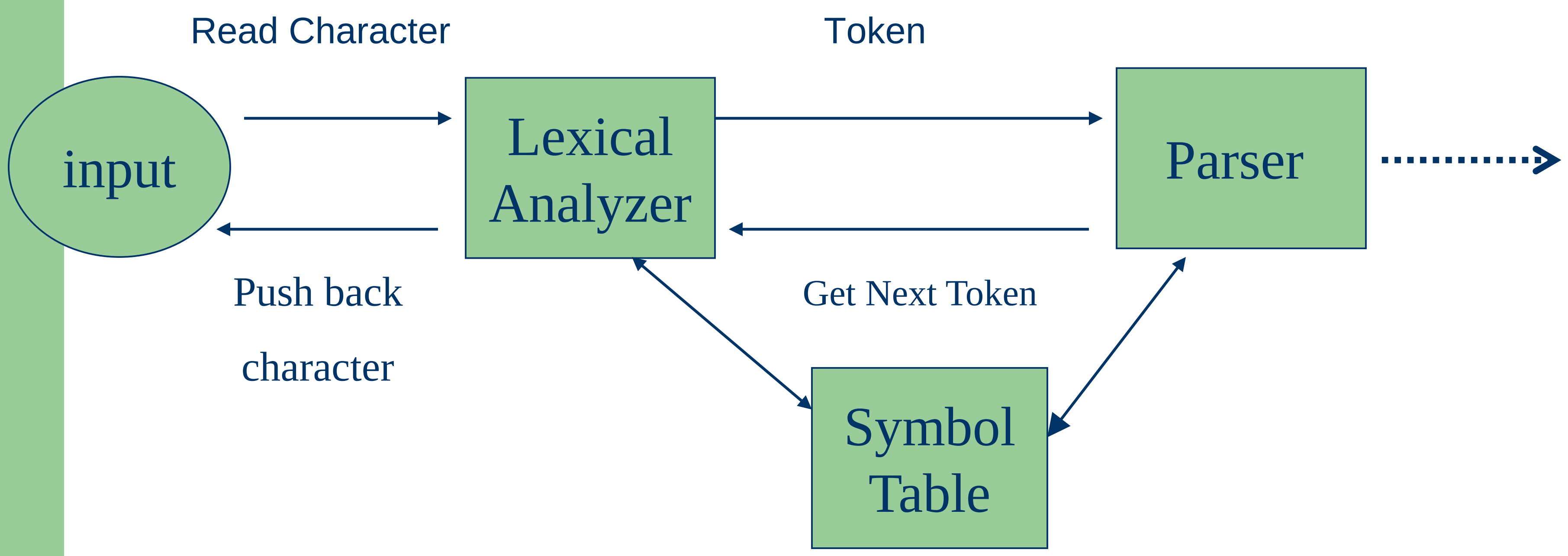


Now!



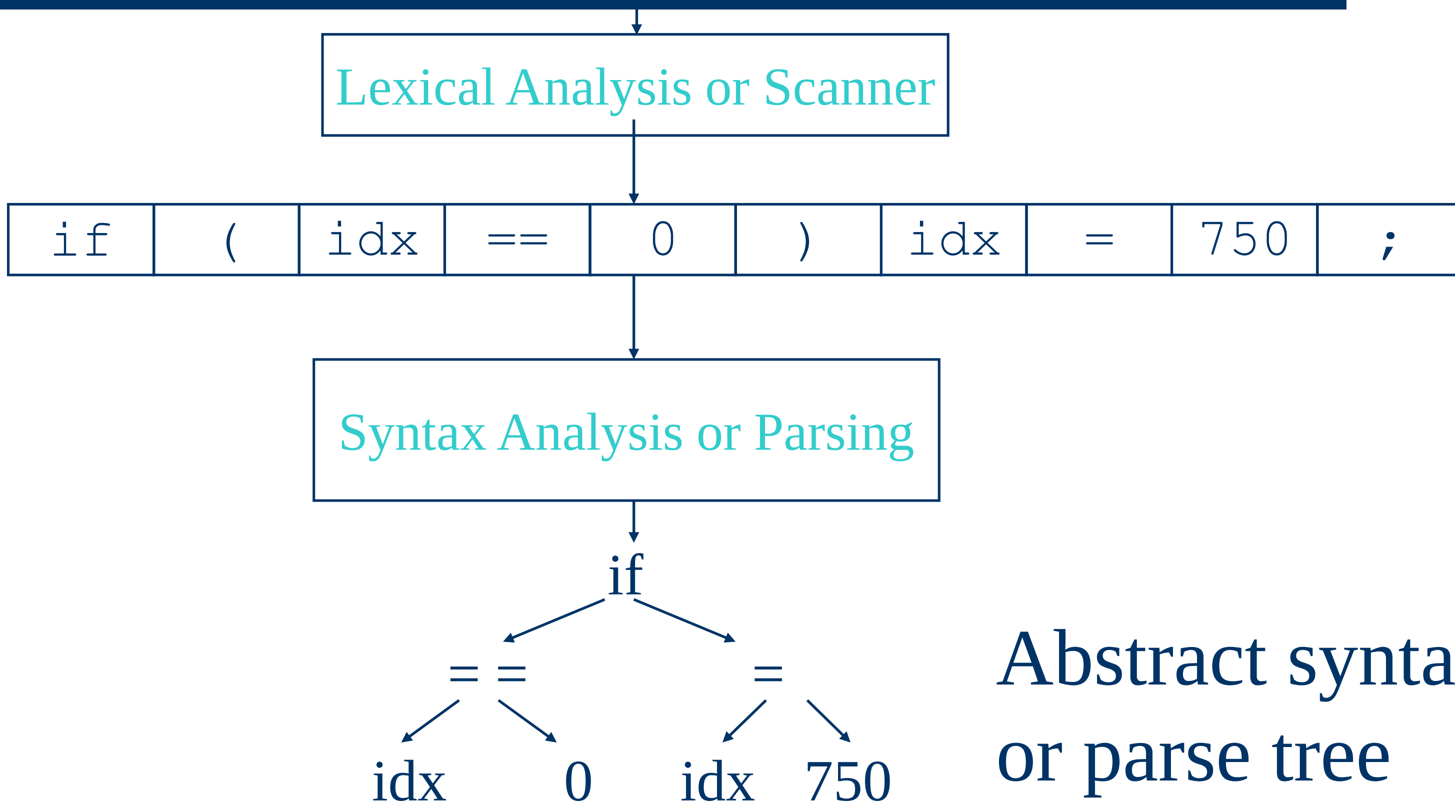
Source Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Int. Code Generator
Front End
Intermediate Code
Code Optimizer
Target Code Generator
Back End
Target Language

THE ROLE OF THE PARSER



Where is Syntax Analysis?

```
if (idx == 0) idx = 750;
```



Abstract syntax tree
or parse tree

Parsing Analogy

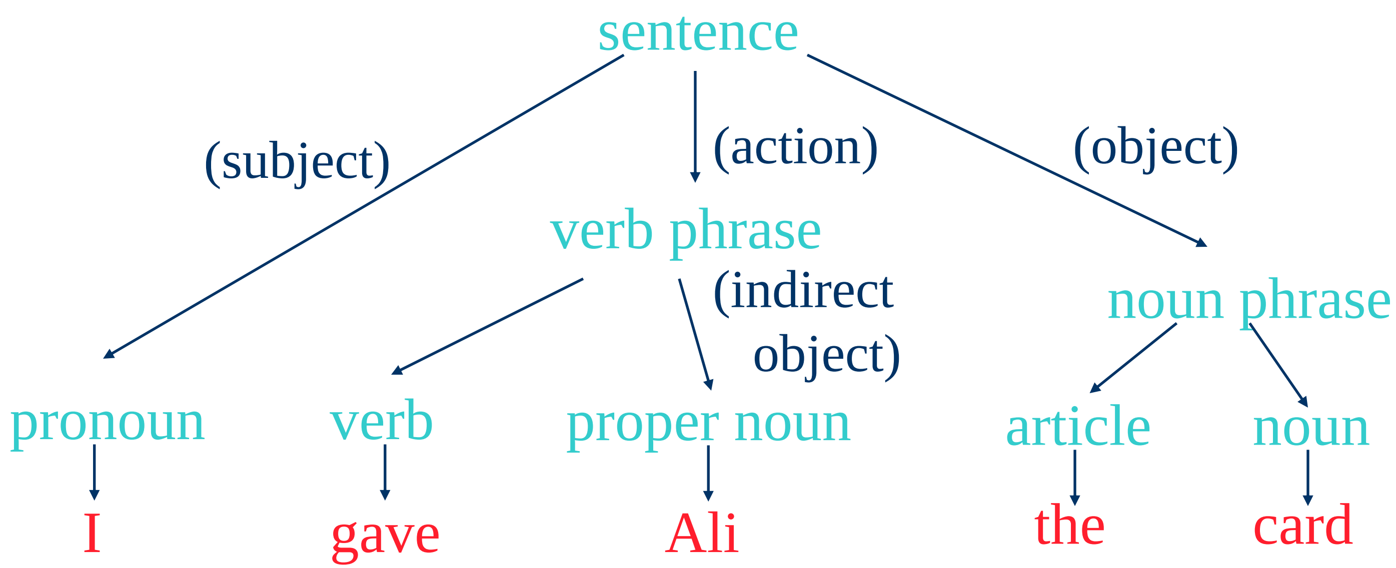
Syntax analysis for natural languages

- Identify the function of each word
- Recognize if a sentence is grammatically correct

Example: I gave Ali the card.

Parsing Analogy

- Syntax analysis for natural languages
  - Identify the function of each word
- Recognize if a sentence is grammatically correct



Syntax Analysis Overview

- Goal: we must determine if the input token stream satisfies the syntax of the program
- What do we need to do this?
  - An expressive way to describe the syntax
  - A mechanism that determines if the input token stream satisfies the syntax description
- For lexical analysis
  - Regular expressions describe tokens
  - Finite automata = mechanisms to generate tokens from input stream

Syntax Analysis(Parsing)

Parsing is the task of determining the syntax of a program. For this reason, it is also called syntax analysis.

The syntax of a programming language is usually given by the grammar rules of a context-free grammar,in a manner similar to the way the lexical structure of the tokens recognized by the scanner is given by the regular expression. Indeed ,a context free grammar uses naming conventions and operations very similar to those of regular expression.

Syntax Analysis(Parsing) (cont'd)

The algorithms used to recognize these structures are also quite different from scanning algorithms. The basic structure used is usually some kind of tree ,called a parse tree or syntax tree.

The Parsing Process

It is the task of the parser to determine the syntactic structure of a programme from the tokens produced by the scanner and either explicitly or implicitly ,to construct a parse tree or syntax tree that represents this structure. Thus the parser may be viewed as a function that takes as its input the sequence of tokens produced by the scanner and produces as its output the syntax tree.

The Parsing Process(cont'd)

Parser

Sequence of tokens $\longrightarrow$ **Syntax Tree**

Usually the sequence of tokens is not an explicit input parameter,but the parser calls a scanner procedure such as getToken to fetch the next token from the input as it is needed during the parsing process.

Context-Free Grammar

We introduce a notation , called a context – free grammar (or grammar) , for specifying the syntax of a language.

A grammar naturally describes the hierarchical structure of many programming language constructs. For example , as if – else statement in C has the from if (expression) statement else statement.

Context-Free Grammar(Cont’d)

That is the statement is concatenation of the keyword an opening parenthesis , an expression , a closing parenthesis , a statement , the keyword else , and another statement. Using the variable expr to denote an expression and the variable stmt to denote a statement , this structuring rule can be expressed as :

Context-Free Grammar(Cont'd)

Stmt $\longrightarrow$ **if (expr) stmt else stmt**

In which the arrow may be reads, a "can have the form ". Such a rule is called **production**.

In a production lexical elements like the keyword if and the parenthesis are called **tokens**.

Context-Free Grammar(Cont’d)

Variables like “expr” and “stmt” represent ,sequences of tokens and called **Nonterminals.**

A context free grammar has four components;

Context-Free Grammar(con't)

- Consist of 4 components (Backus-Naur Form or BNF):

1) A set of tokens , known as **termin**al symbol
2) A set of **non terminals**.
3) A set of **productions** where each production consists of non-terminals , called the left side of the production , an arrow and a sequence of tokens and for non-terminals called right side of the production.
4) A designation of one of the non-terminals as the **starts symbol**.

Backus Naur Form (BNF)

BNF stands for **Backus Naur Form** notation understood as Backus Naur Formas introduced by John Bakus and Peter Naur in 1960.

1. It is metasyntax notation for context-free grammars.
2. It is a formal method for describing the syntax of programming language.
3. It is used to write a formal representation of a context-free grammar.

Backus Naur Form (BNF)

Different languages have different description and rules but the general structure of BNF is given as:

```
name ::= expansion
```

- The symbol ::= means "may expand into" and "may get replaced with."
- Every name in Backus-Naur form is surrounded by angle brackets, < >, whether it appears on the left- or right-hand side of the rule.
- An expansion is an expression containing terminal symbols and non-terminal symbols, joined together by sequencing and selection.

- A terminal symbol may be a literal like ("+" or "function") or a category of literals (like integer).
- A vertical bar | indicates choice.

Example of Production of Grammar

There is the production for any grammar as follows:

$$S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow c$$

In BNF, we can represent above grammar as follows:

$$S \rightarrow aSa \mid bSb \mid c$$

BNF EXAMPLE

BNF e.g:

```
<expr> ::= <term> "+" <expr>
        |  <term>

<term> ::= <factor> "*" <term>
        |  <factor>

<factor> ::= "(" <expr> ")"
          |  <const>

<const> ::= integer
```

Context-Free Grammar(con't)

EXAMPLE:

expr $\longrightarrow$ expr op expr

expr $\longrightarrow$ (expr)

expr $\longrightarrow$ id

op $\longrightarrow$ +

op $\longrightarrow$ -

op $\longrightarrow$ *

Context-Free Grammar(cont'd)

- Terminal Symbols :

  id , + , -, *, (,)
- Non-Terminal Symbols:

  expr,op
- Start Symbol

  expr
- Production

  $\text{expr} \longrightarrow \text{expr op expr}$

Example 1

We use expressions consisting of digits and plus and minus signs,

e.g. **9 – 5+2**, since a plus or minus sign appear between two digits. We refer to such expressions as lists of digits separated by plus or minus sign expressions. The following grammar describe the syntax of these expressions.

Example 1(Cont'd)

The productions are:

List $\longrightarrow$ list + digits (1)

List $\longrightarrow$ list – digits (2)

List $\longrightarrow$ digit (3)

Digit $\longrightarrow$ 0,1,2,3,4,5,6,7,8,9

Example 1(Cont’d)

The right sides of the productions with non terminals list on the left side can equivalently be grouped;

List $\longrightarrow$ list + digit|list – digit | digit

The token of the grammar are the symbol are the symbols + - 0123456789.

The non terminals are list and digit, with list being the starting non terminals because its production are given first .

Example 1(Cont’d)

We say a production is for a non terminal if the non terminals appears on the left side of the production . A string of tokens is sequence of zero or more tokens. The string containing zero tokens , written as “ε” is called the empty string.

Example 1(Cont’d)

The language defined by the grammar of example 1, consists list of digits separated by plus and minus signs. We can deduce that

9-5+2 is a list as follows.

Example 1(Cont'd)

- 9 is a list by production (3), since 9 is a digit
- 9-5 is a list by production (2) , since 9 is a list and 2 is a digit
- 9-5 +2 is a list by production (1) , since 9-5 is a list and 2 is a digit.

Example 1(Cont'd)

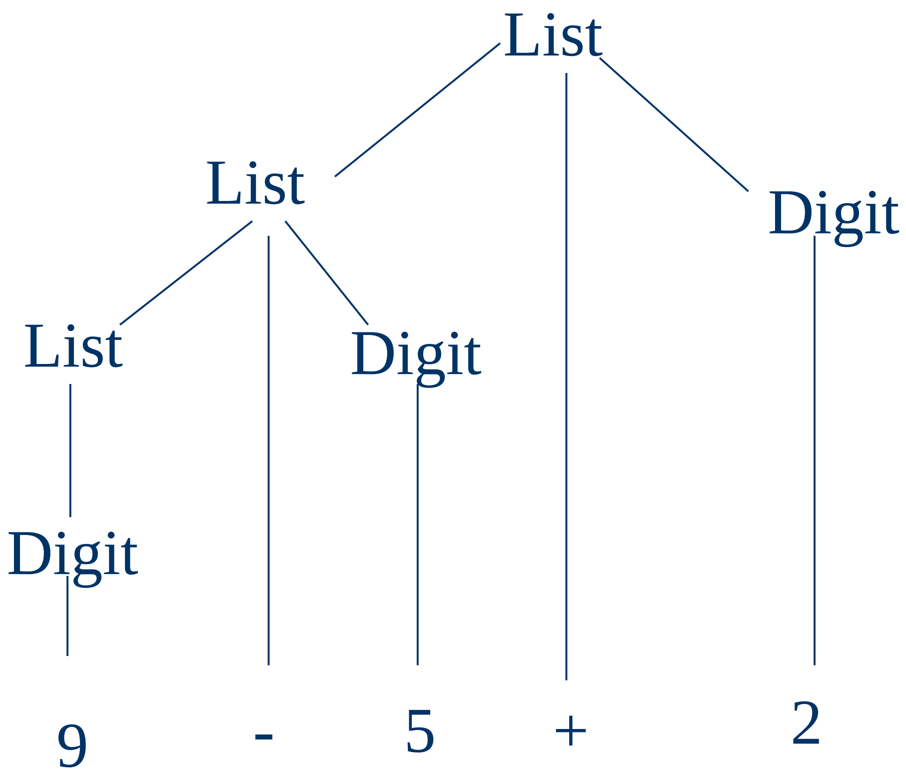
This reasoning is shown by the tree in next slide . Each node in the tree is labeled by a grammar symbol .An interior node and its children correspond to a production : the interior node corresponds to the left side of the production ,the children to the right side.

Such trees are called parse trees.

List = list + digit

Interior node Children

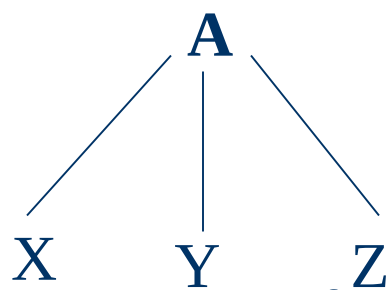
Example 1(Cont'd)



Parse Tree

A parse tree shows how the start symbol of a grammar derives a string in the language. If non terminal A has production A $\longrightarrow$ XYZ, then a parse tree may have an interior node labeled **A** with three children labeled **X**, **Y** and **Z**, from left to right.

Parse Tree(Cont'd)



Formally , given a context free grammar , a parse tree is a tree with the following properties;

Defining a Parse Tree

- More Formally, a Parse Tree for a CFG Has the Following Properties:
  - Root Is Labeled With the Start Symbol
  - Leaf Node Is a Token or $\in$
  - Interior Node (Now Leaf) Is a Non-Terminal
  - If $A \rightarrow x1x2\ldots xn$, Then A Is an Interior; $x1x2\ldots xn$ Are Children of A and May Be Non-Terminals or Tokens

Ambiguity

If a grammar can have more than one parse tree generating a given string of tokens , then such a grammar is said to be **ambiguous** to show that a grammar is ambiguous all we need to do is find a token string that has more then one parse tree. Since a string with more then one parse tree usually has more than one meaning for compiling applications we need to design unambiguous grammars.

Ambiguity (Cont'd)

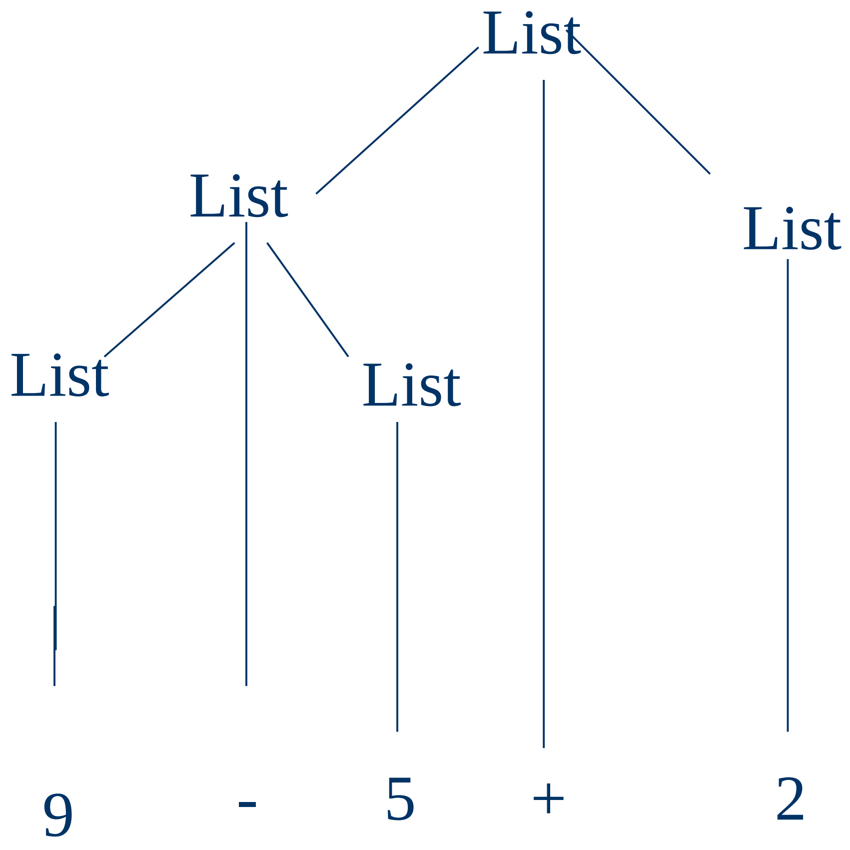
Suppose we did not distinguish between digits and lists as in example (1). We could have written the grammar.

$$List \longrightarrow list + list$$

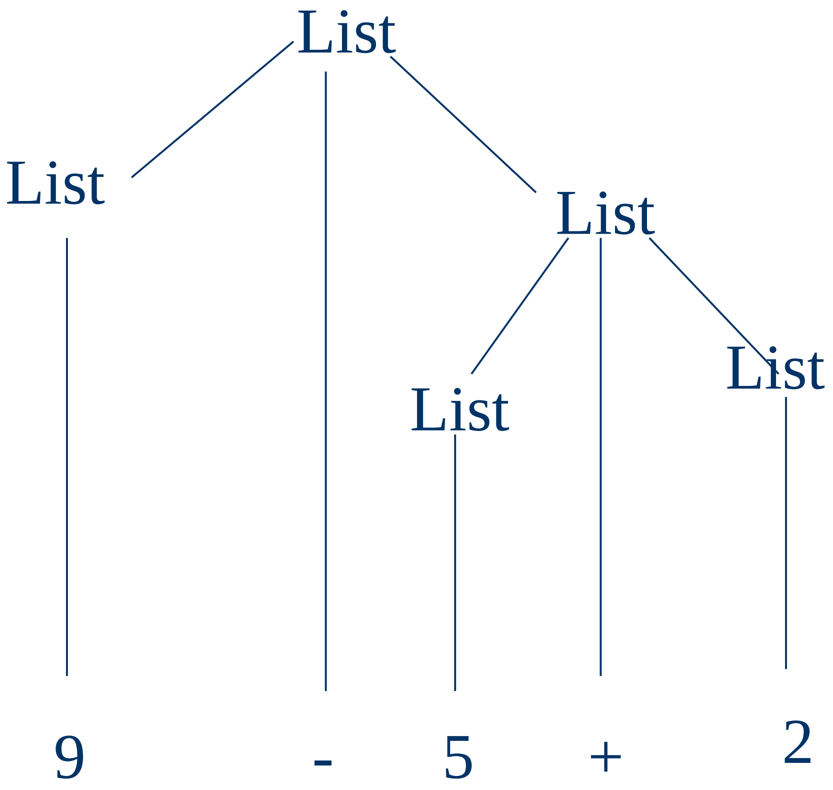
$$List \longrightarrow list - list$$

$$List \longrightarrow 0|1|2|3|4|5|6|7|8|9$$

Ambiguity (Cont'd)



Ambiguity (Cont'd)



True Derivation

Op = '+'|'-'|'*'|'/'

Int = [0-9]+

Open = (

Close =)

1) *Start* → *Expr*

2) *Expr* → *Expr* Op *Expr*

3) *Expr* → Int

4) *Expr* → Open *Expr* Close

Start

Expr

Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1

Parse Tree Construction

1) $Start \rightarrow Expr$

2) $Expr \rightarrow Expr$ Op $Expr$

3) $Expr \rightarrow$ Int

4) $Expr \rightarrow$ Open $Expr$ Close

Start

Start

Expr

Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1

Parse Tree Construction

1) *Start* $\rightarrow$ *Expr*

2) *Expr* $\rightarrow$ *Expr* Op *Expr*

3) *Expr* $\rightarrow$ Int

4) *Expr* $\rightarrow$ Open *Expr* Close

Start

Start

Expr

Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1

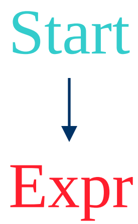
Parse Tree Construction

1) $Start \rightarrow Expr$

2) $Expr \rightarrow Expr$ Op $Expr$

3) $Expr \rightarrow$ Int

4) $Expr \rightarrow$ Open $Expr$ Close



Start

Expr

Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1

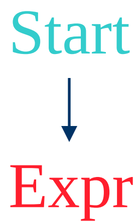
Parse Tree Construction

1) *Start* → *Expr*

2) *Expr* → *Expr* Op *Expr*

3) *Expr* → Int

4) *Expr* → Open *Expr* Close



Start

Expr

Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1

Parse Tree Construction

1) $\mathit{Start} \rightarrow \mathit{Expr}$

2) $\mathit{Expr} \rightarrow \mathit{Expr}\ \mathrm{Op}\ \mathit{Expr}$

3) $\mathit{Expr} \rightarrow \mathrm{Int}$

4) $\mathit{Expr} \rightarrow \mathrm{Open}\ \mathit{Expr}\ \mathrm{Close}$

Start

Expr

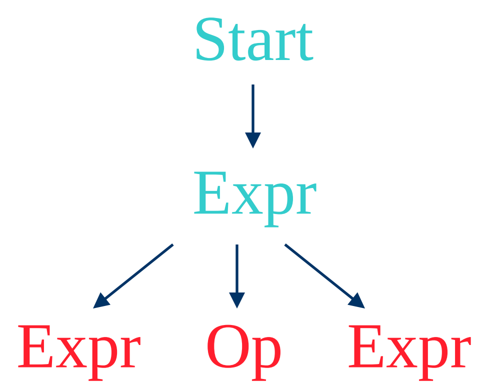
Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1



Parse Tree Construction

1) *Start* → *Expr*

2) *Expr* → *Expr* Op *Expr*

3) *Expr* → Int

4) *Expr* → Open *Expr* Close

Start

Expr

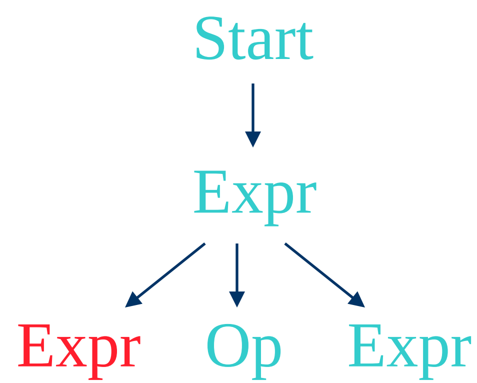
Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1



Parse Tree Construction

1) $Start \rightarrow Expr$

2) $Expr \rightarrow Expr$ Op $Expr$

3) $Expr \rightarrow$ Int

4) $Expr \rightarrow$ Open $Expr$ Close

Start

Expr

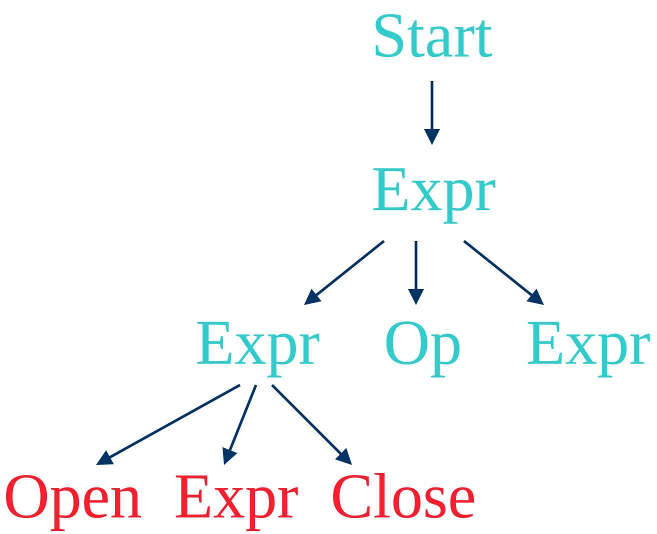
Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

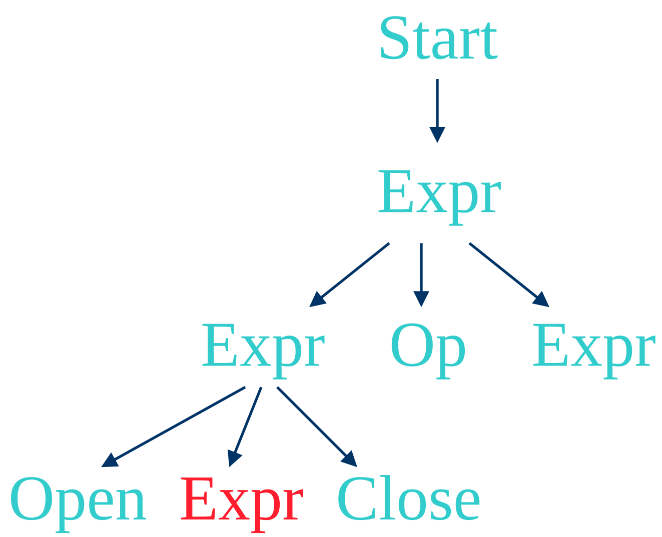
(2 - 1) + 1



Parse Tree Construction

1) *Start* → *Expr*
2) *Expr* → *Expr* Op *Expr*
3) *Expr* → Int
4) *Expr* → Open *Expr* Close

Start
Expr
Expr Op *Expr*
Open *Expr* Close Op *Expr*
Open *Expr* Op *Expr* Close Op *Expr*
Open Int Op Int Close Op Int
(2 - 1) + 1



Parse Tree Construction

1) $Start \rightarrow Expr$

2) $Expr \rightarrow Expr$ Op $Expr$

3) $Expr \rightarrow$ Int

4) $Expr \rightarrow$ Open $Expr$ Close

Start

Expr

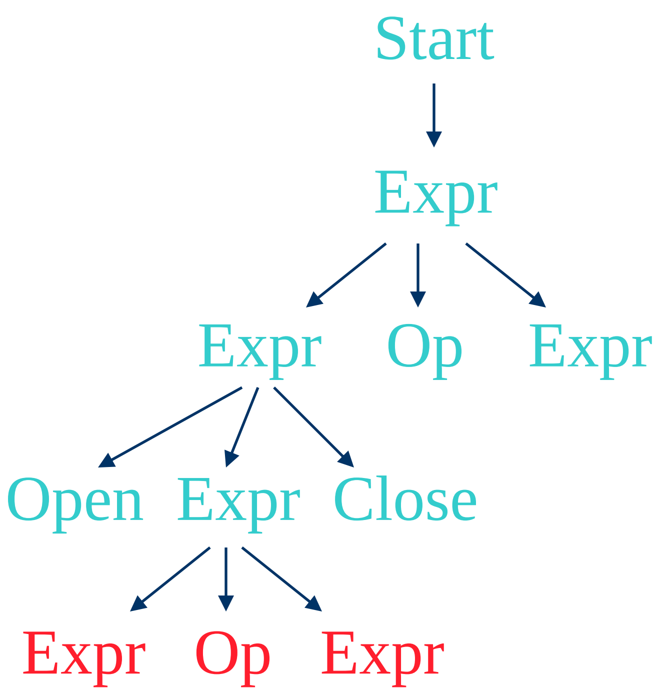
Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1



Parse Tree Construction

1) *Start* → *Expr*

2) *Expr* → *Expr* Op *Expr*

3) *Expr* → Int

4) *Expr* → Open *Expr* Close

Start

Expr

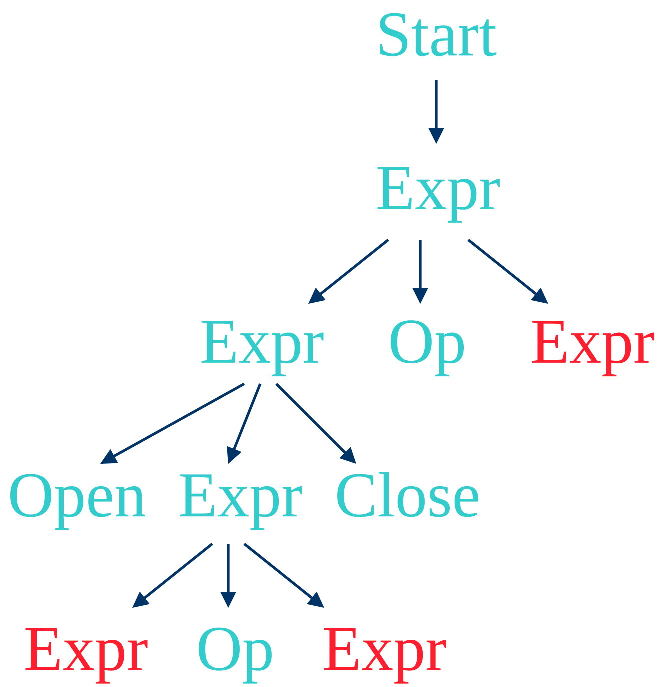
Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1



Parse Tree Construction

1) *Start* $\rightarrow$ *Expr*

2) *Expr* $\rightarrow$ *Expr* Op *Expr*

3) *Expr* $\rightarrow$ Int

4) *Expr* $\rightarrow$ Open *Expr* Close

Start

Expr

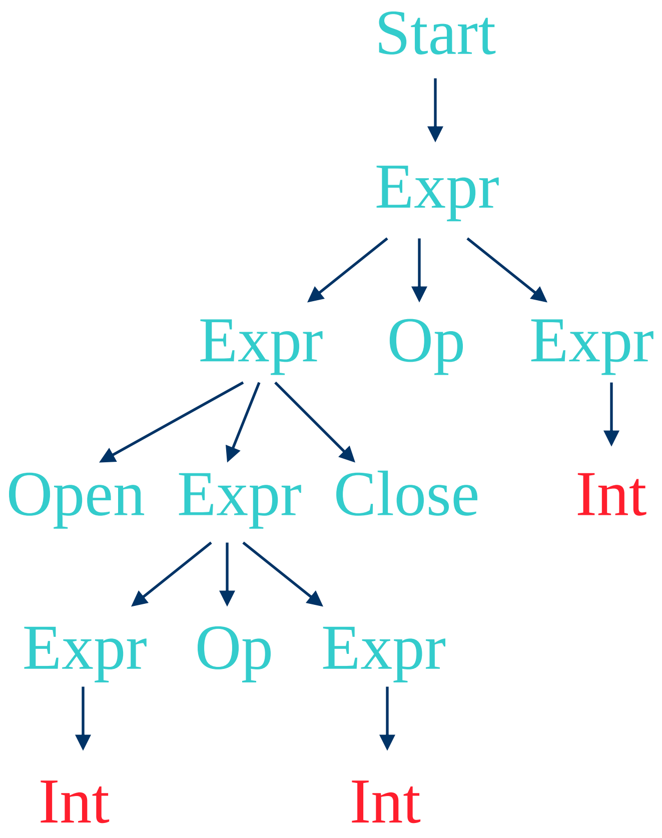
Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

Open Int Op Int Close Op Int

(2 - 1) + 1



Parse Tree Construction

1) $Start \rightarrow Expr$

2) $Expr \rightarrow Expr$ Op $Expr$

3) $Expr \rightarrow$ Int

4) $Expr \rightarrow$ Open $Expr$ Close

Start

Expr

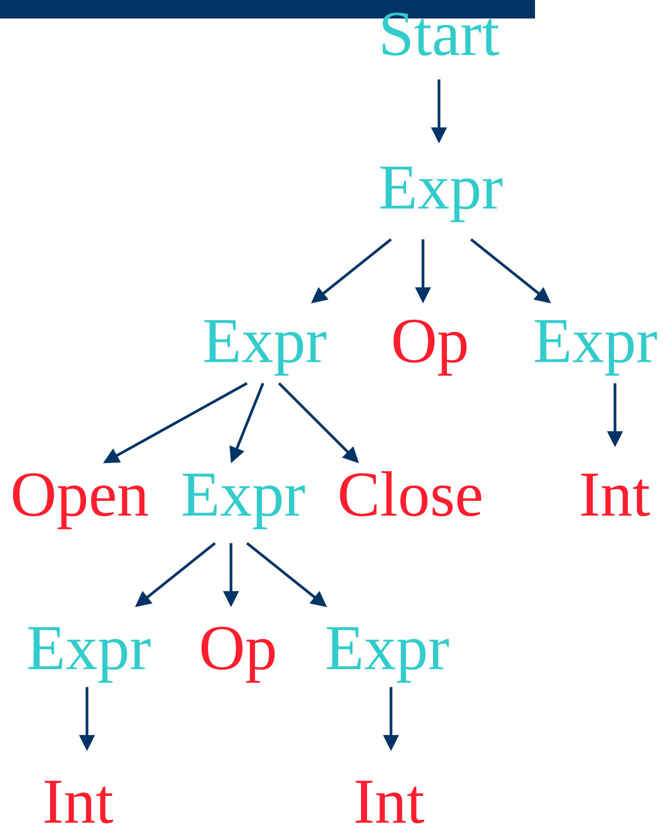
Expr Op *Expr*

Open *Expr* Close Op *Expr*

Open *Expr* Op *Expr* Close Op *Expr*

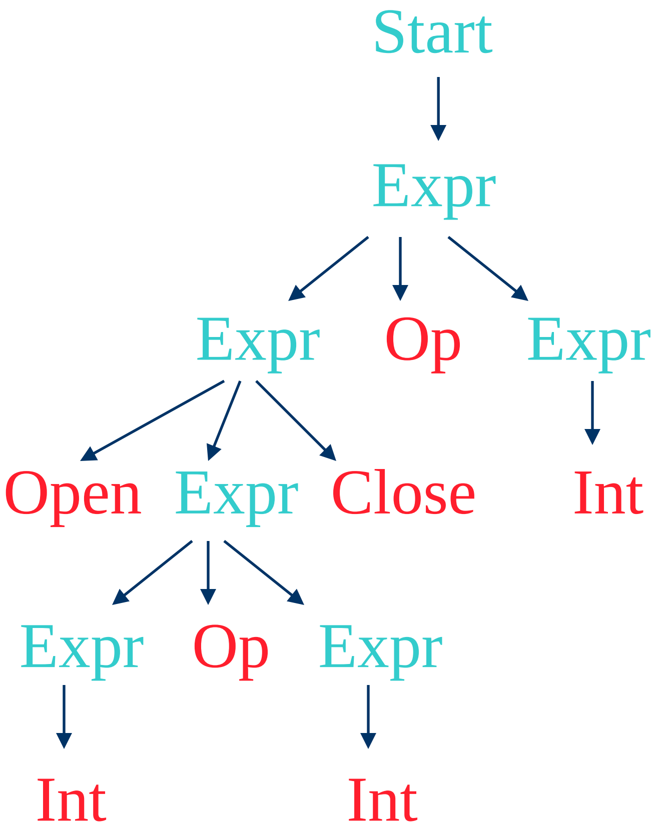
Open Int Op Int Close Op Int

(2 - 1) + 1

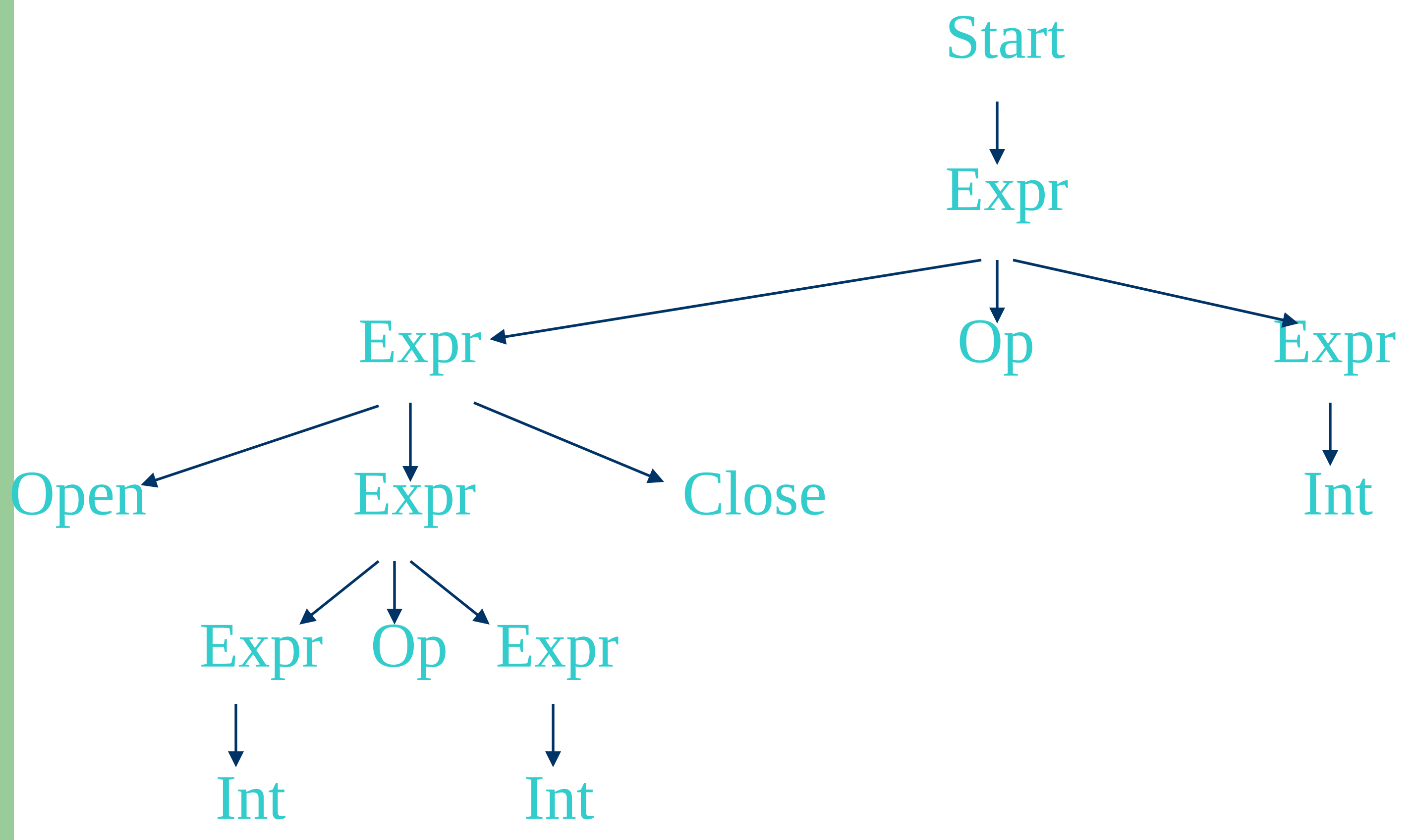


DONE!

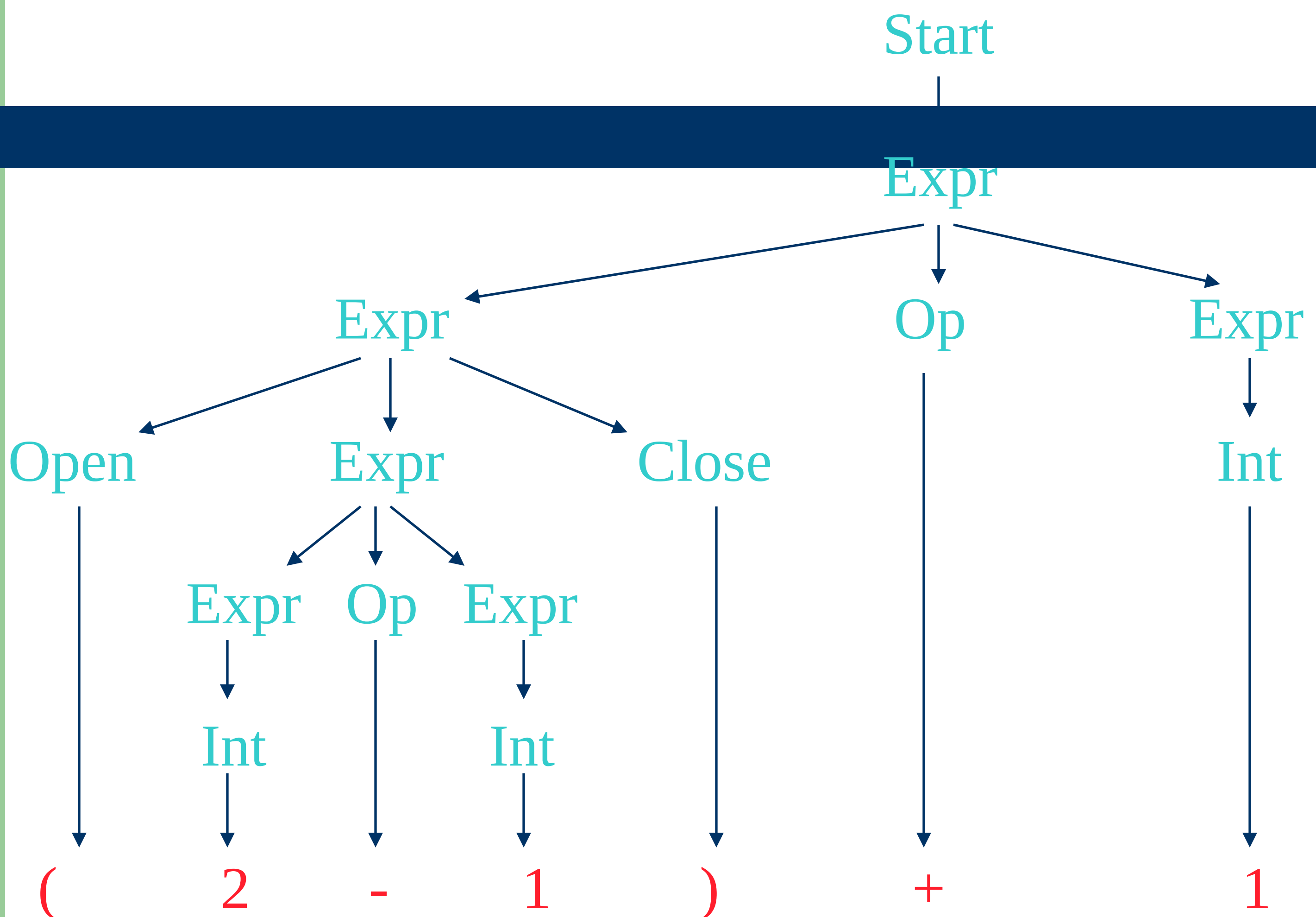
Processing the Tree



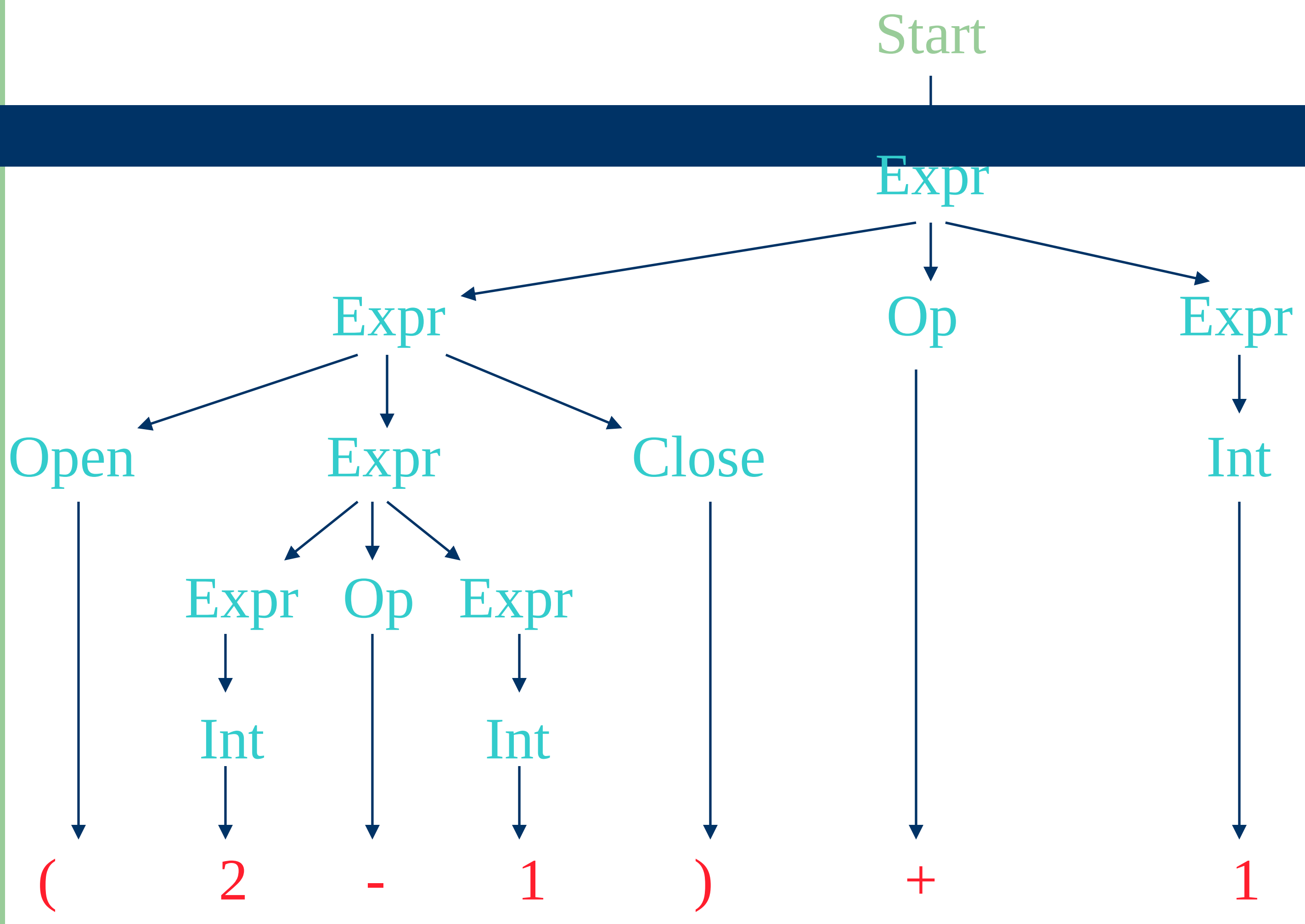
Processing the Tree



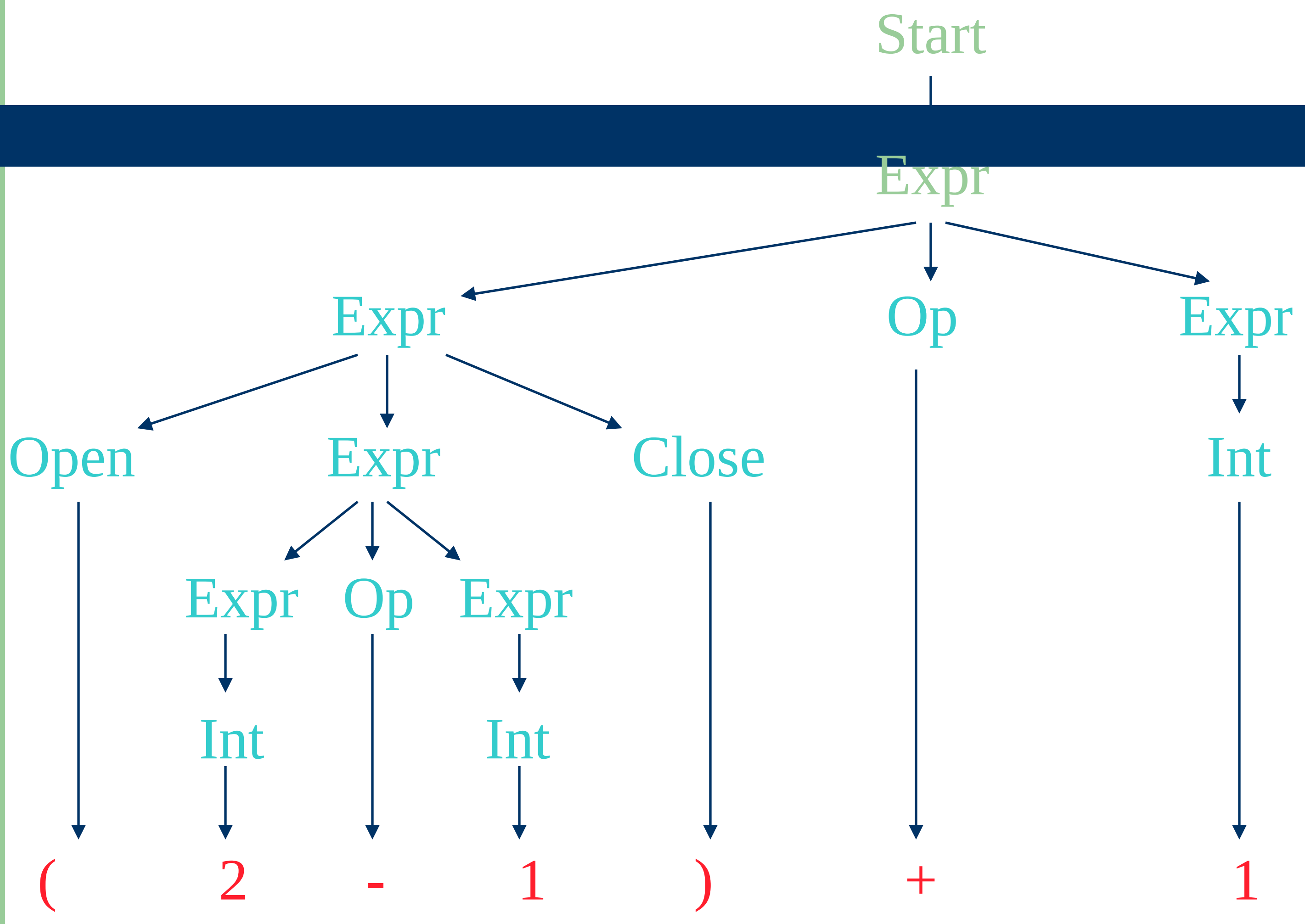
Processing the Tree



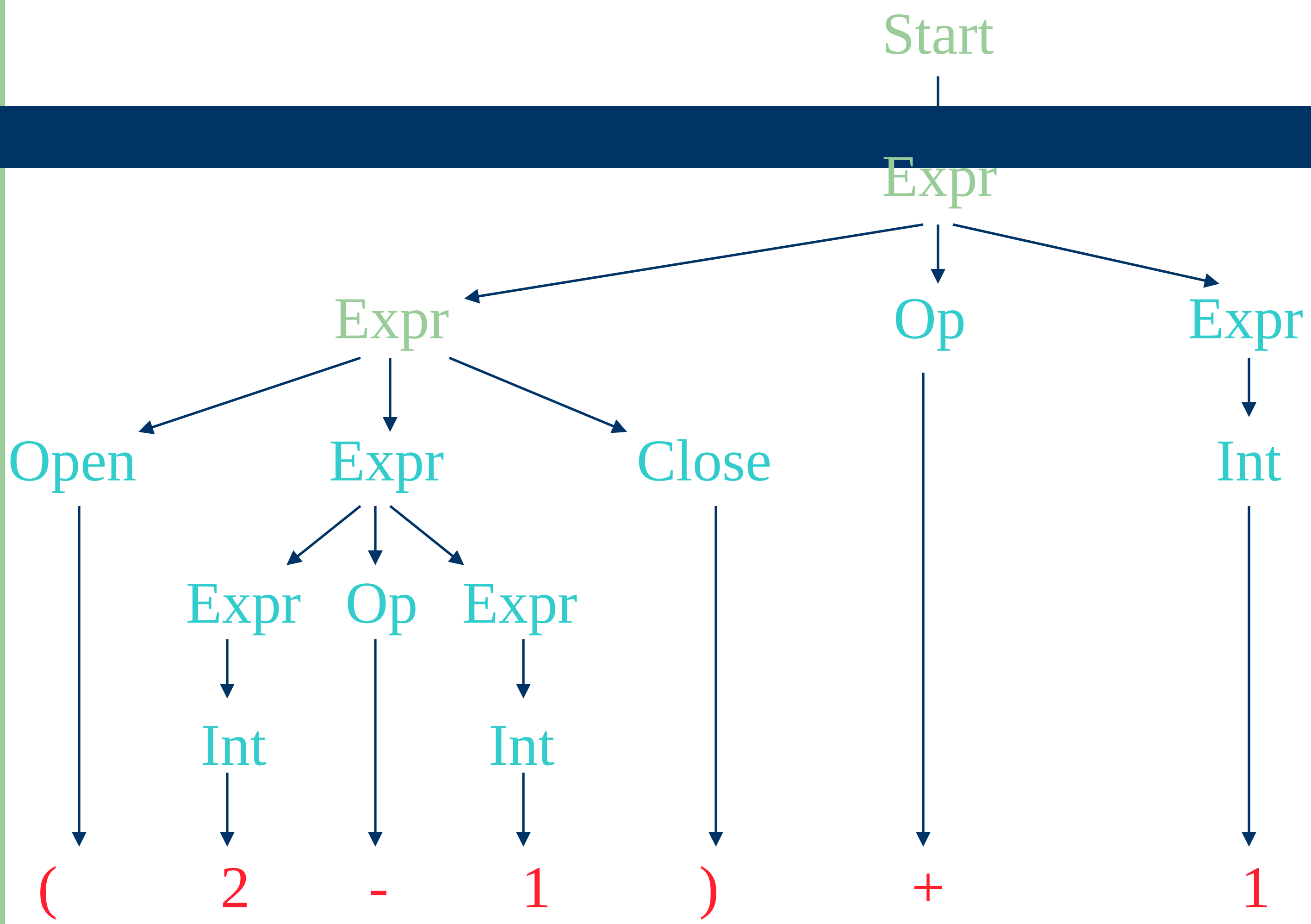
Processing the Tree



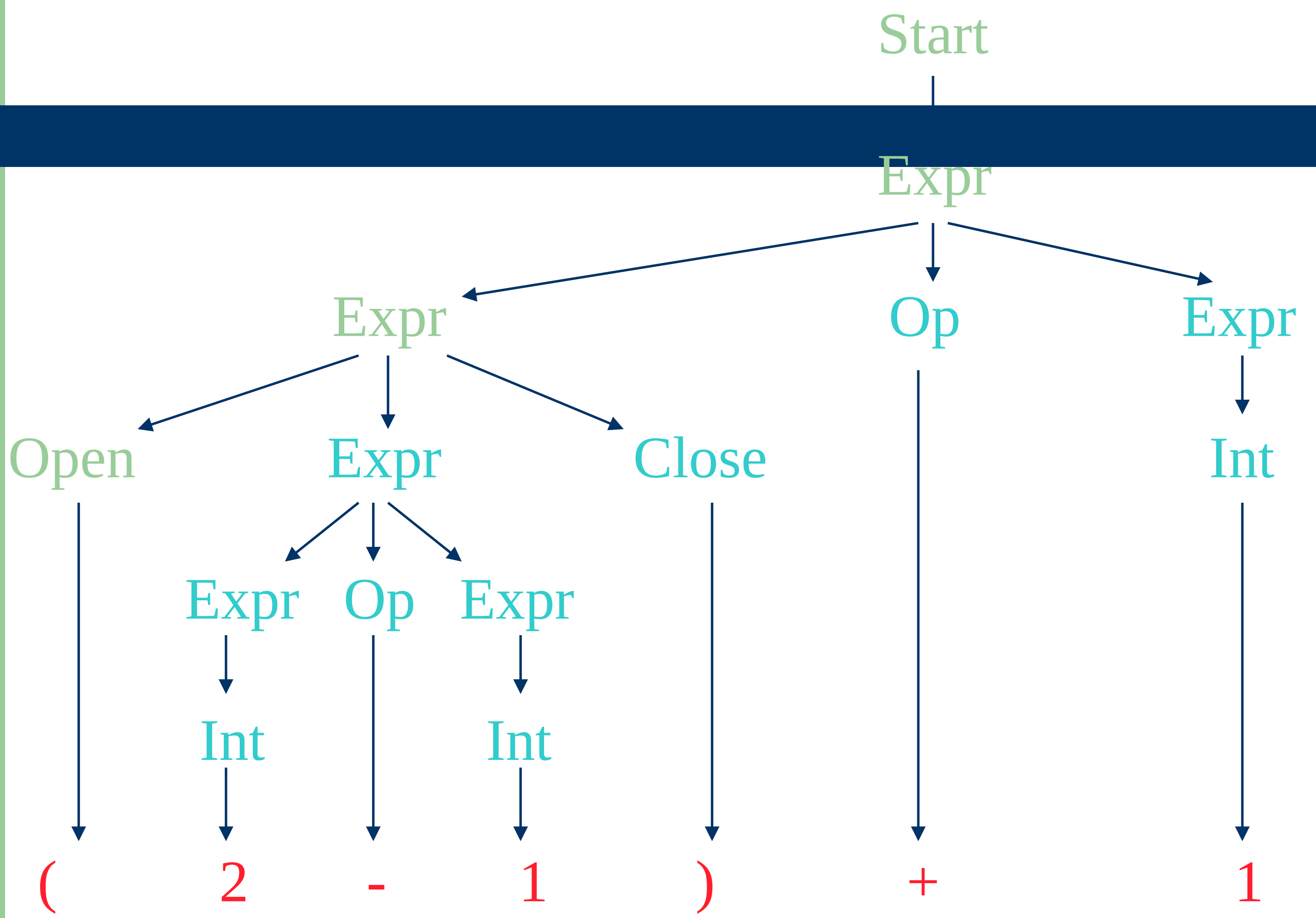
Processing the Tree



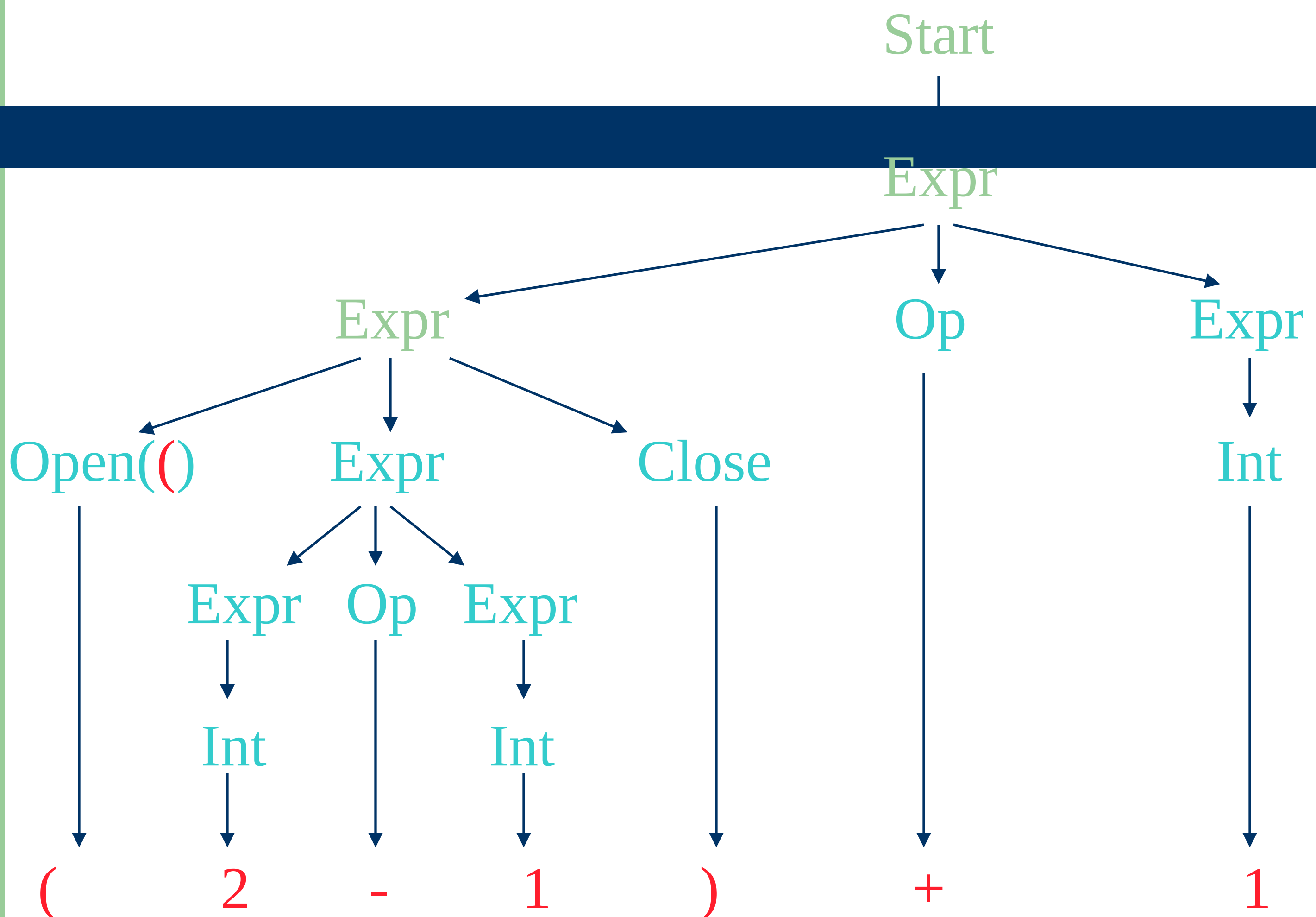
Processing the Tree



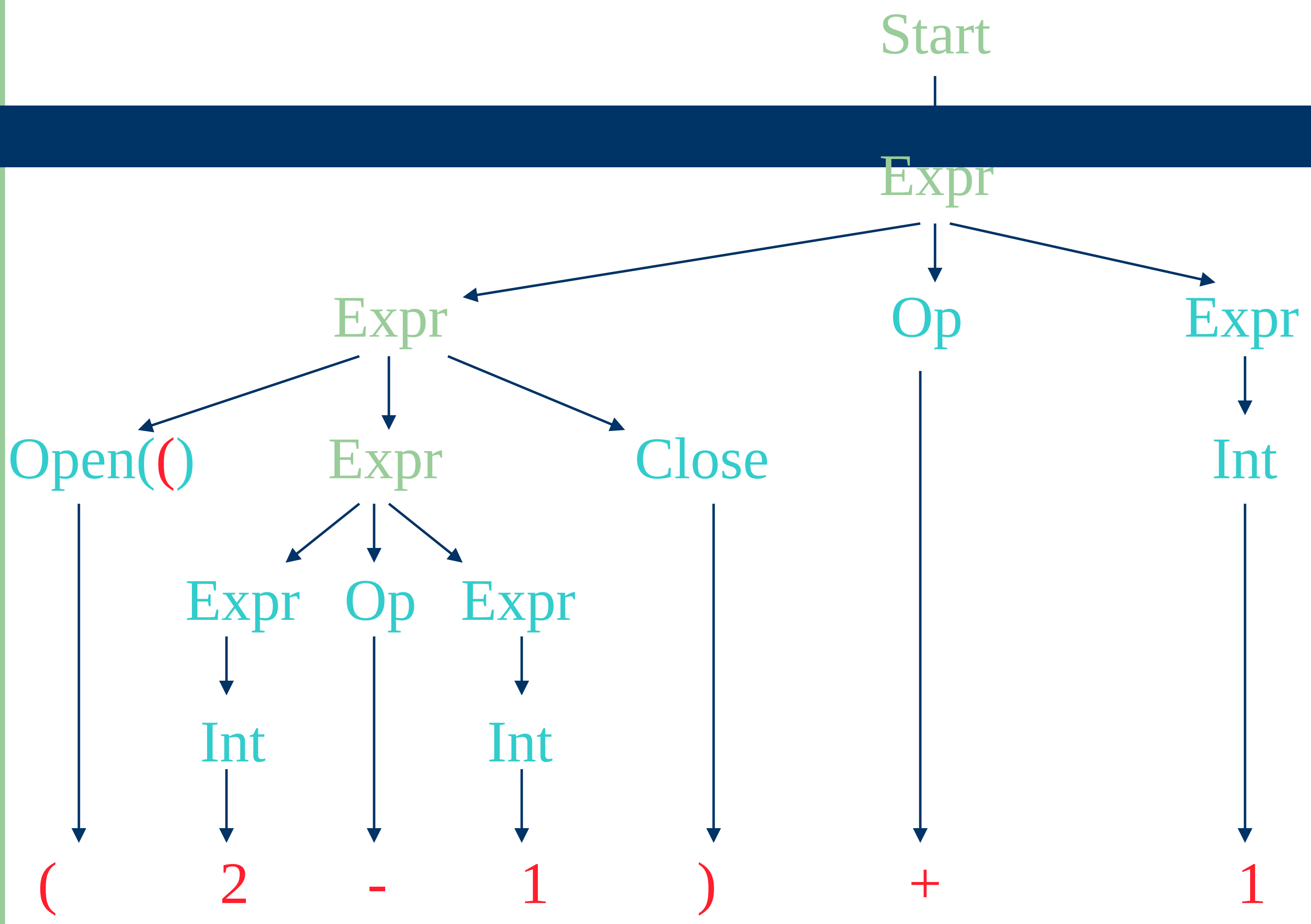
Processing the Tree



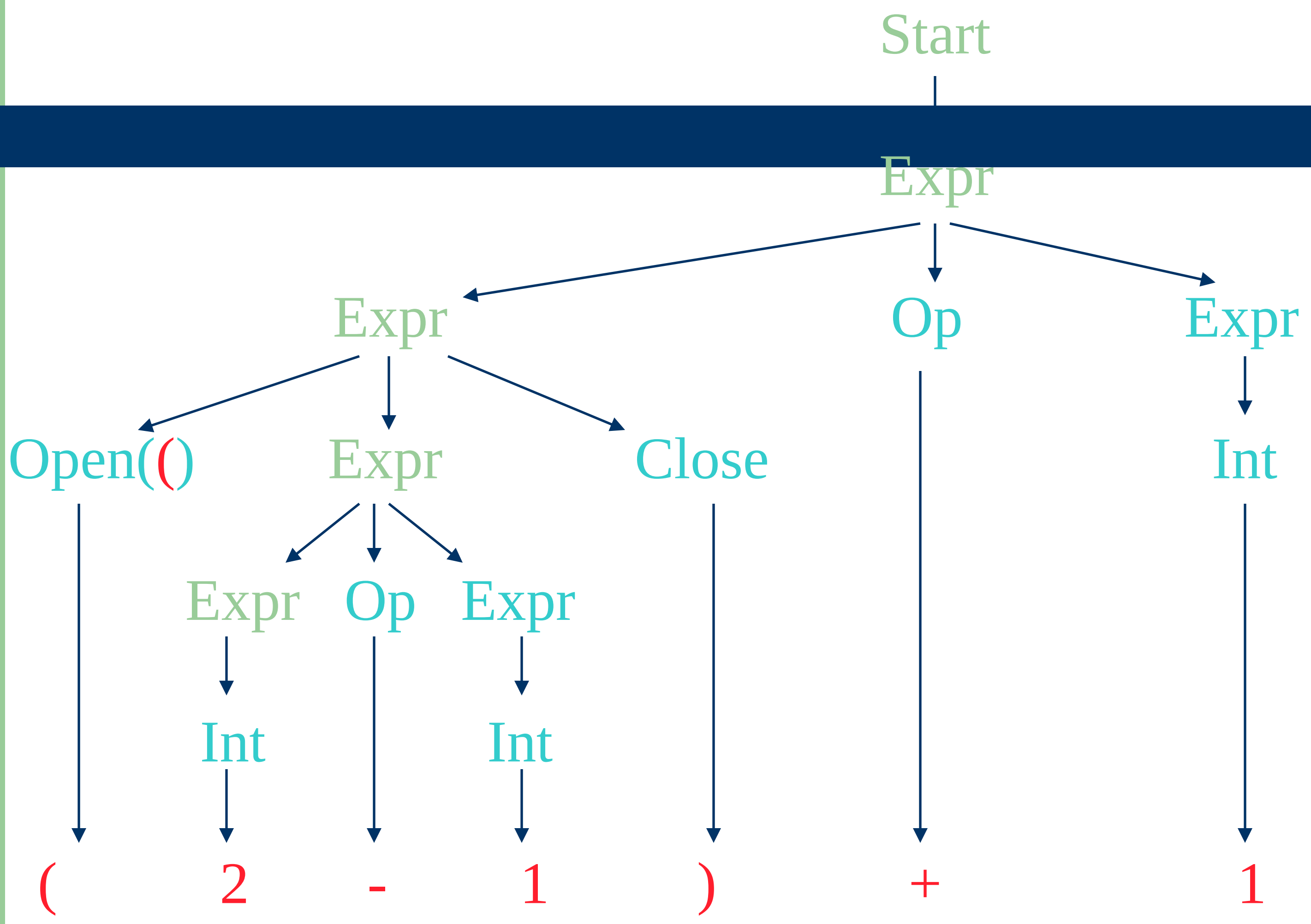
Processing the Tree



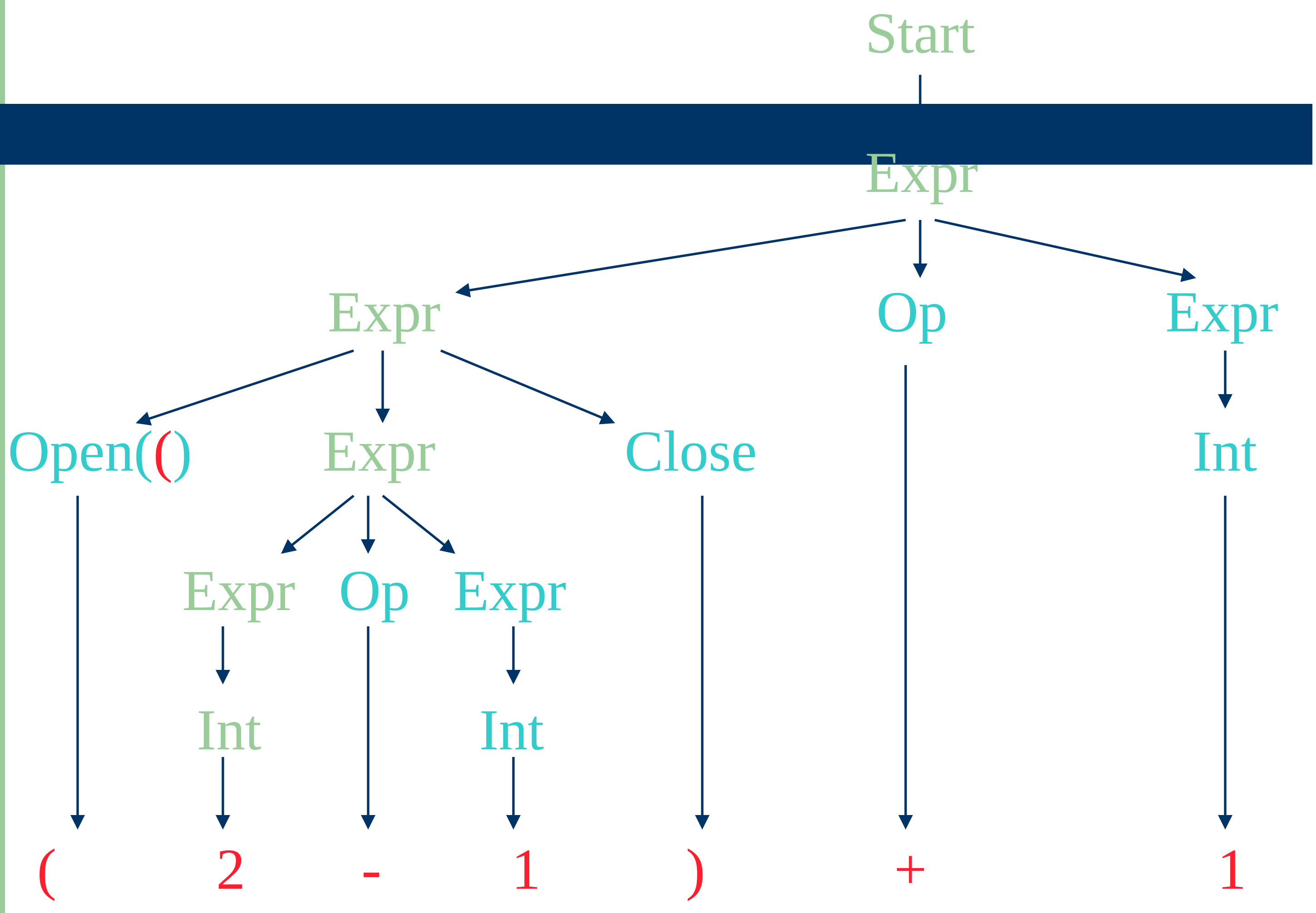
Processing the Tree



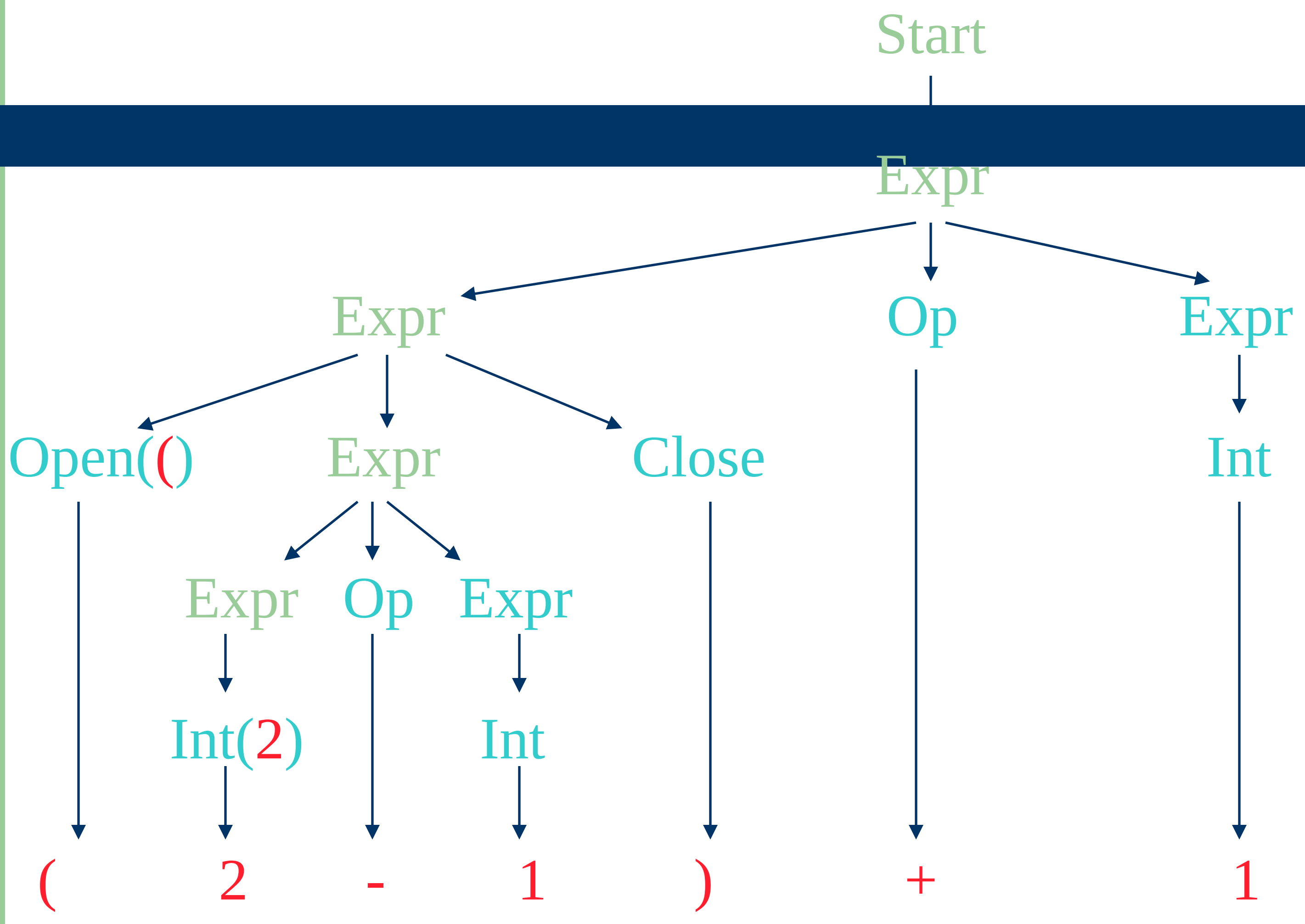
Processing the Tree



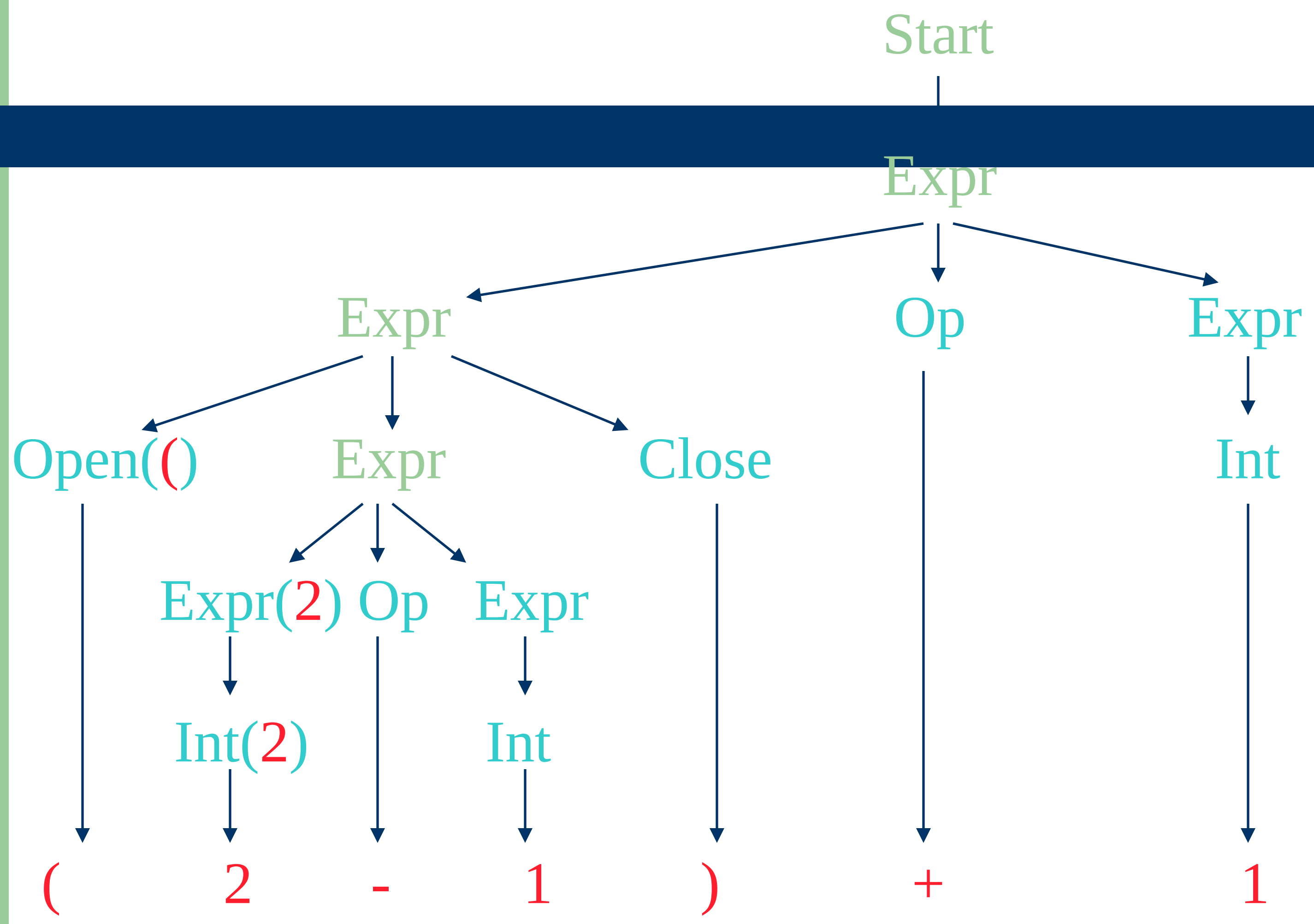
Processing the Tree



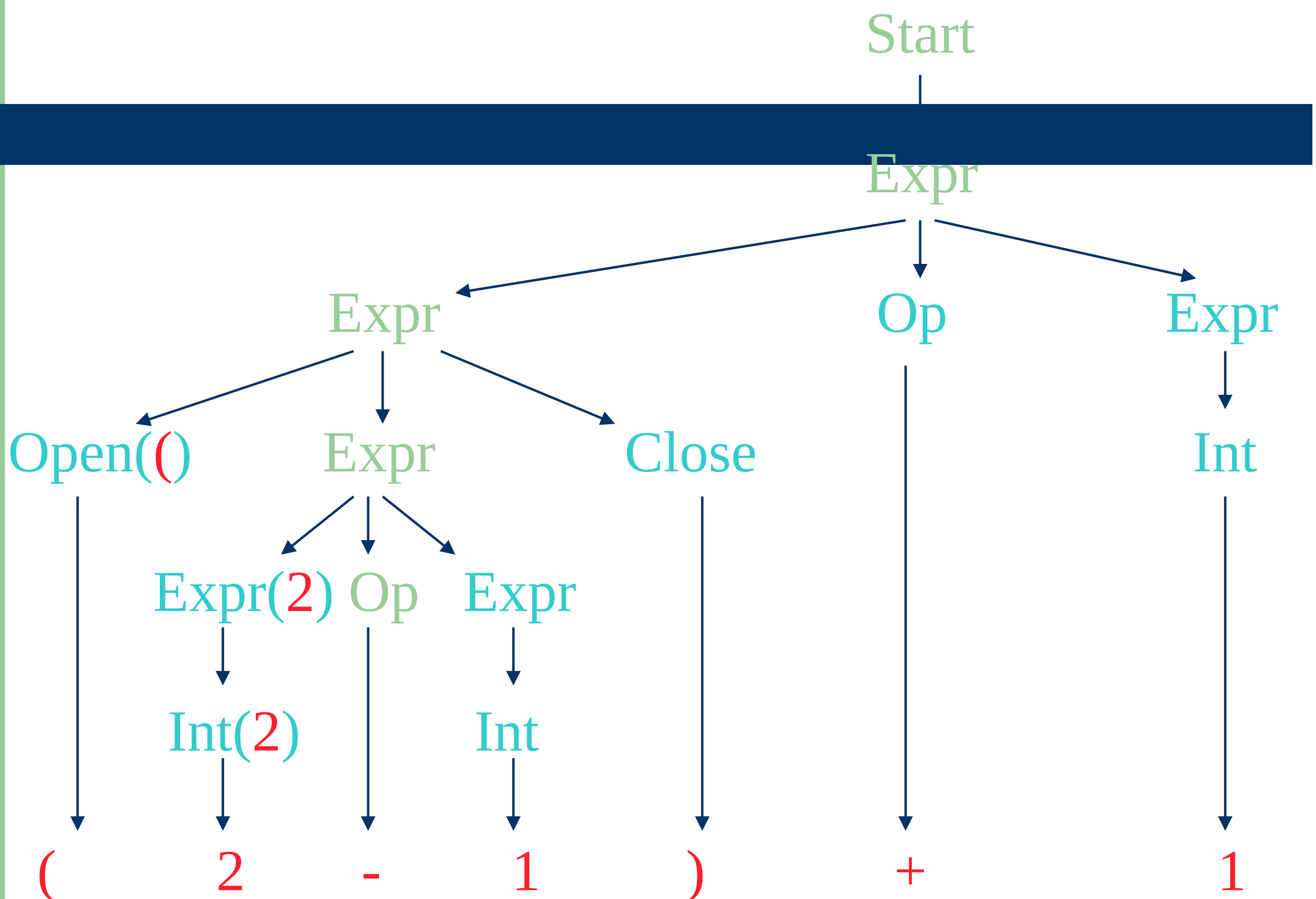
Processing the Tree



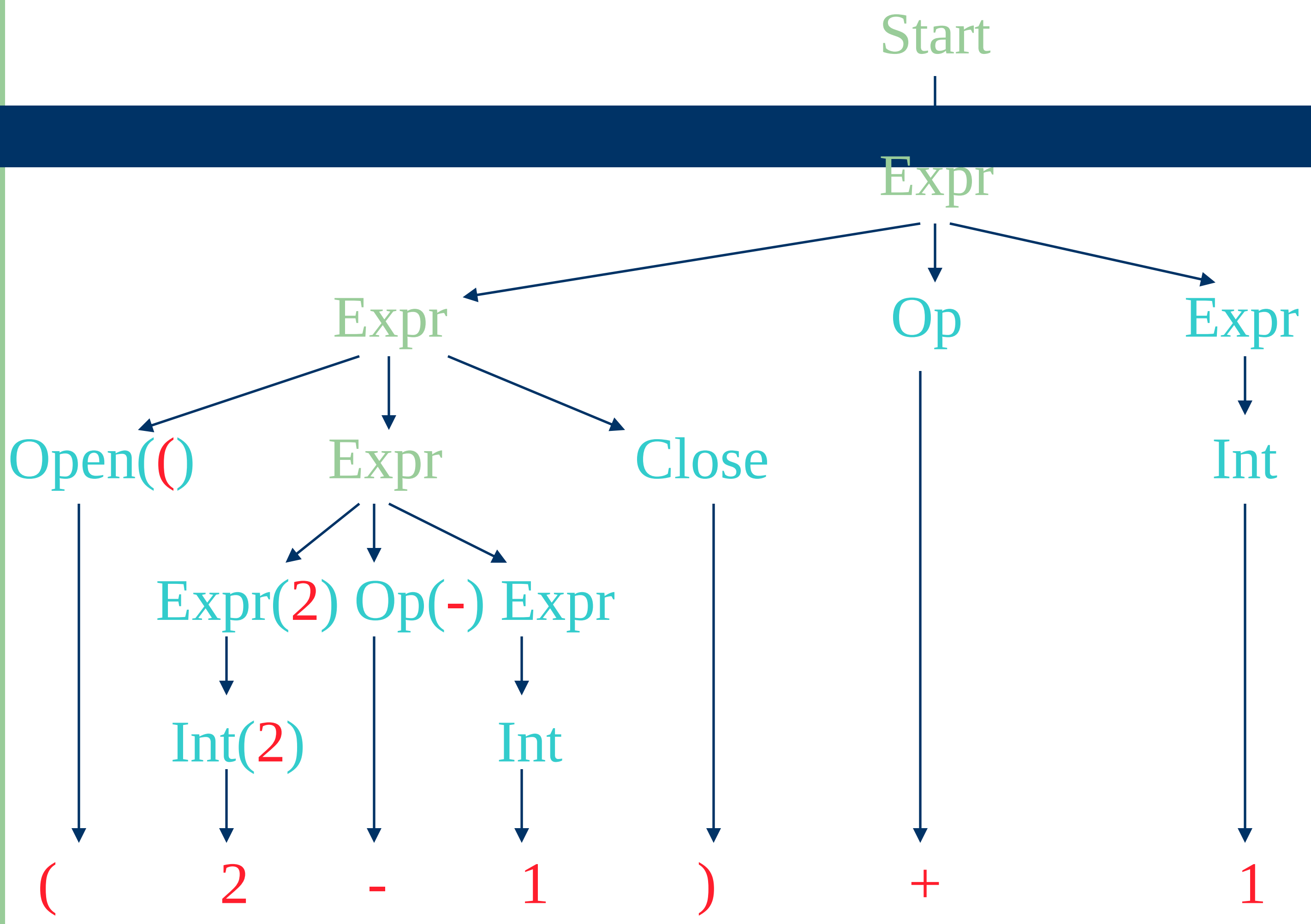
Processing the Tree



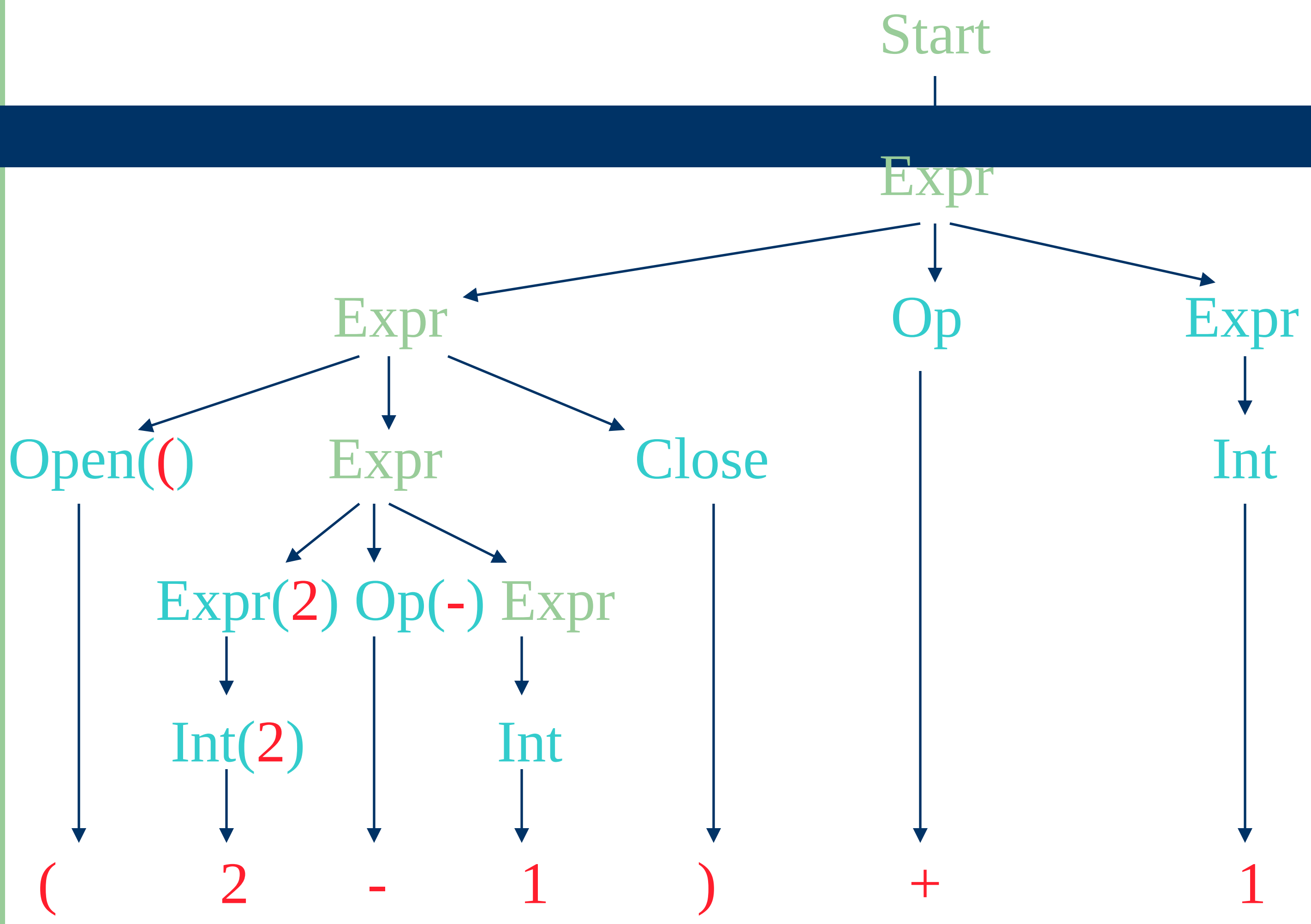
Processing the Tree



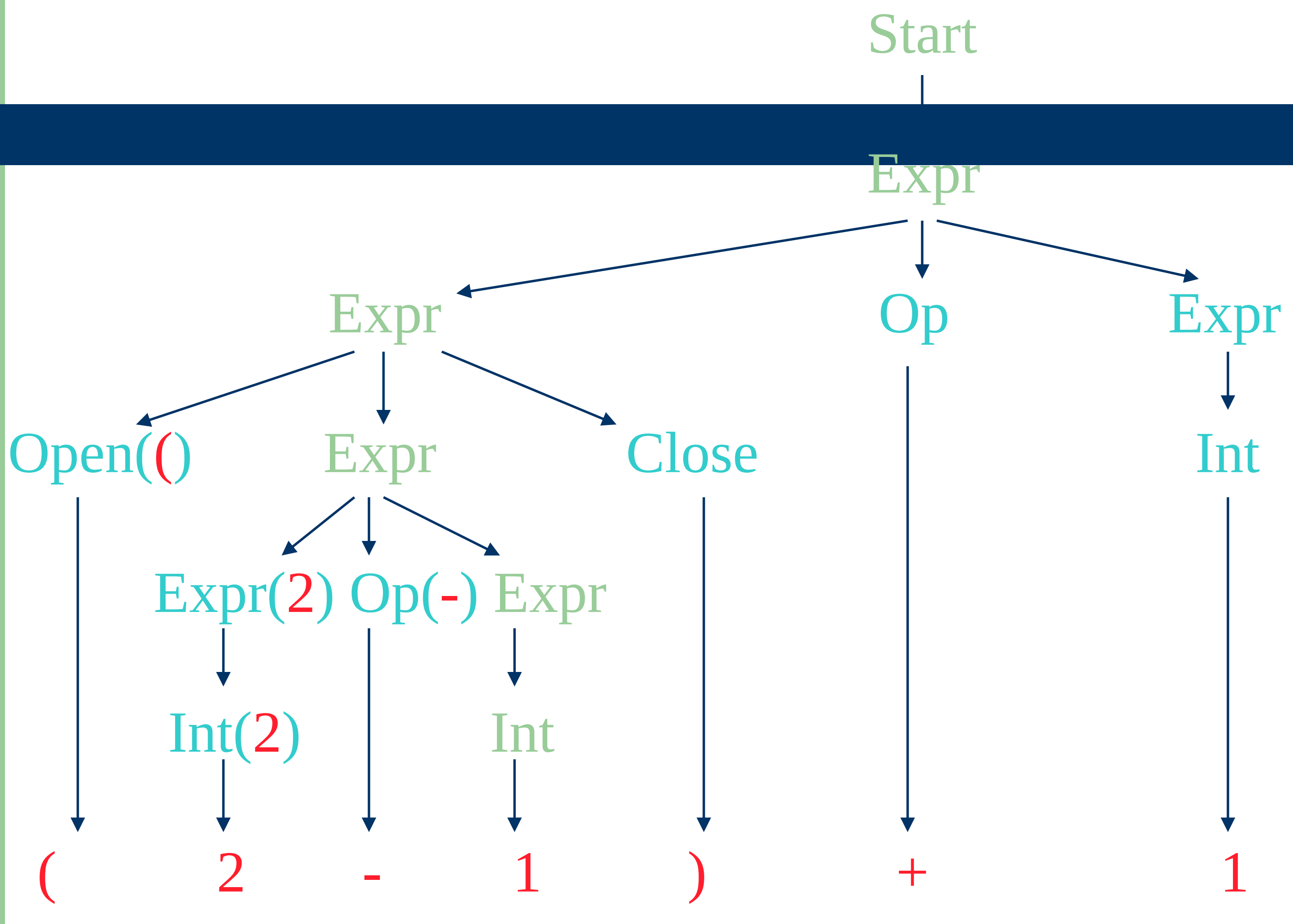
Processing the Tree



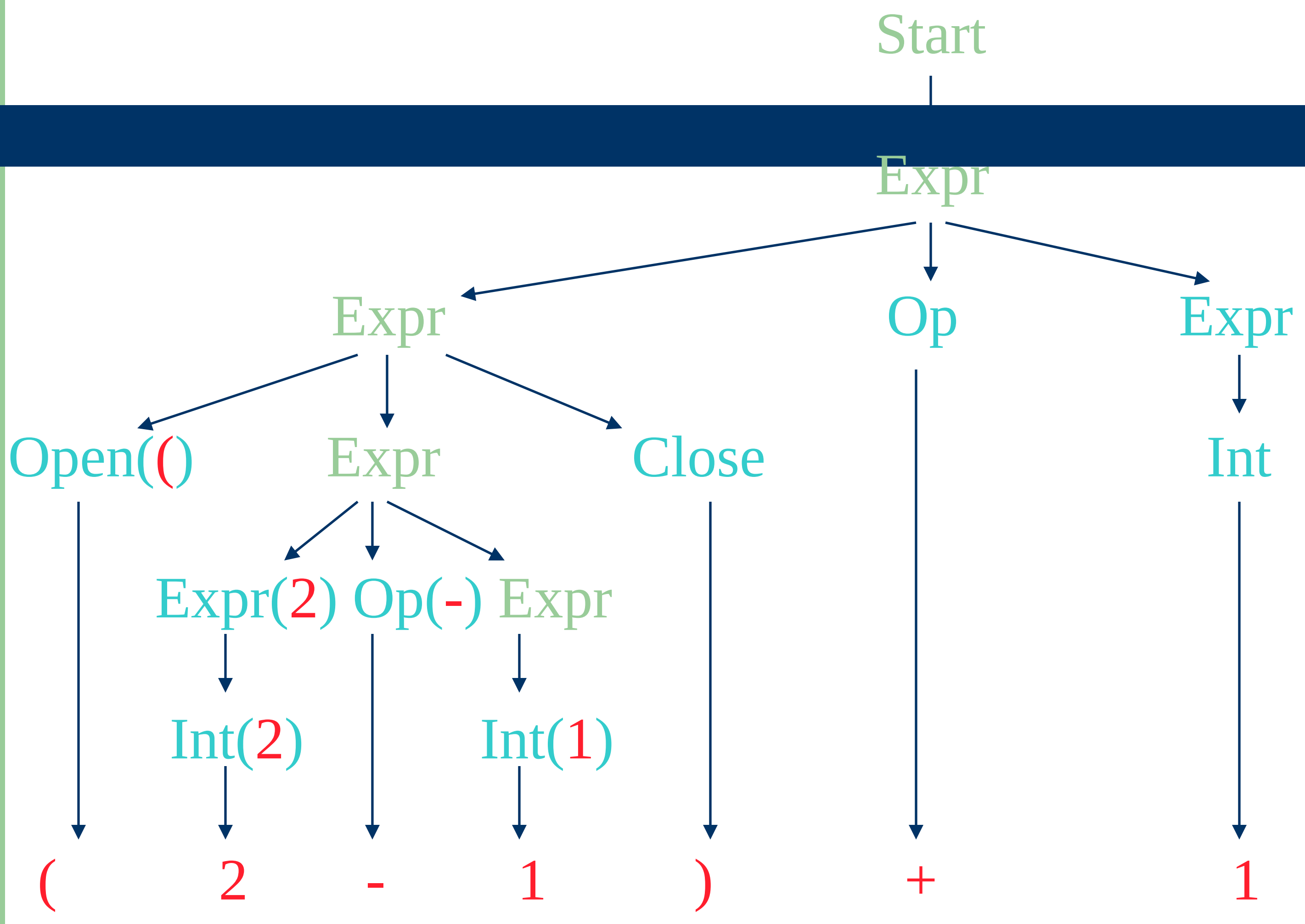
Processing the Tree



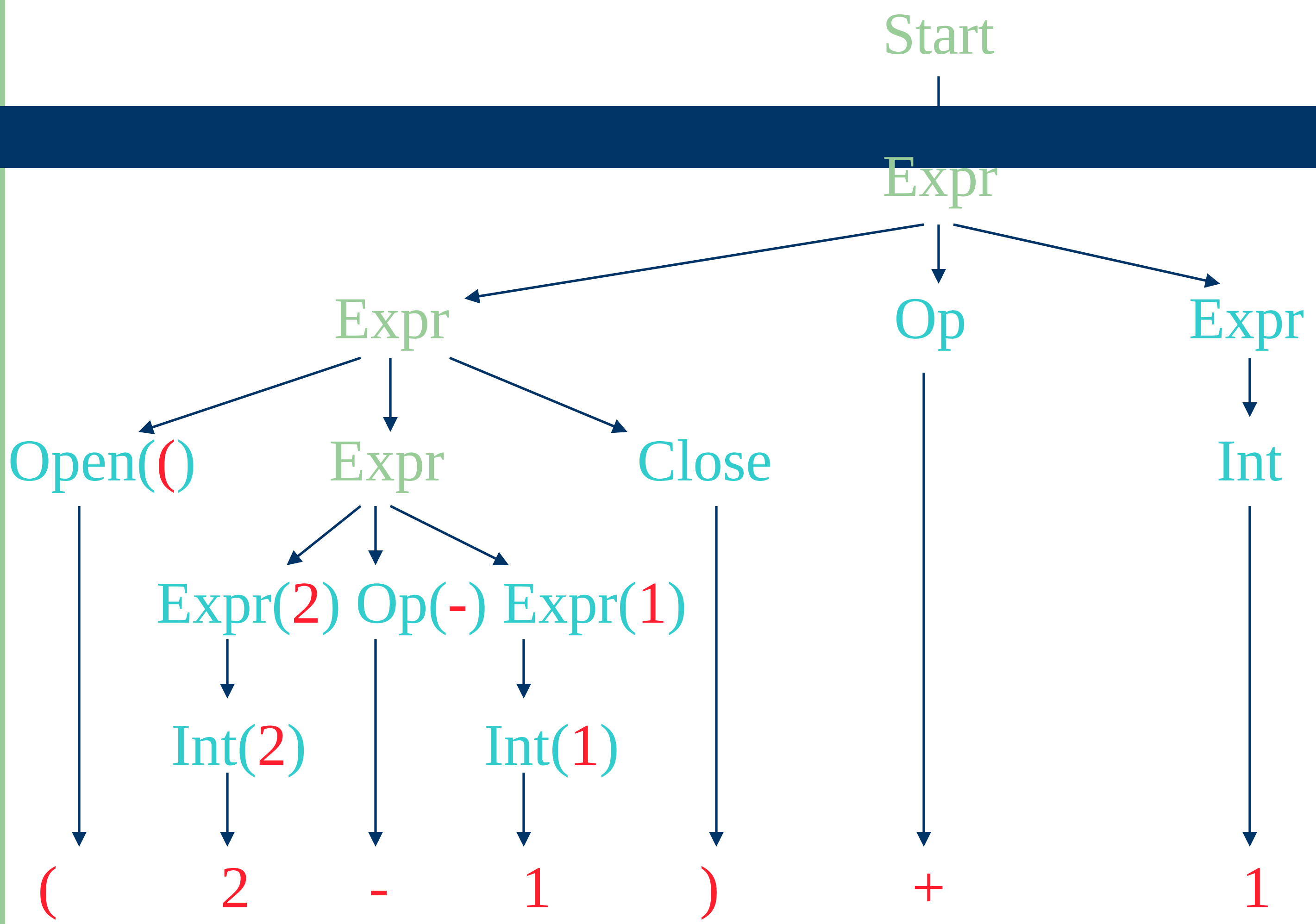
Processing the Tree



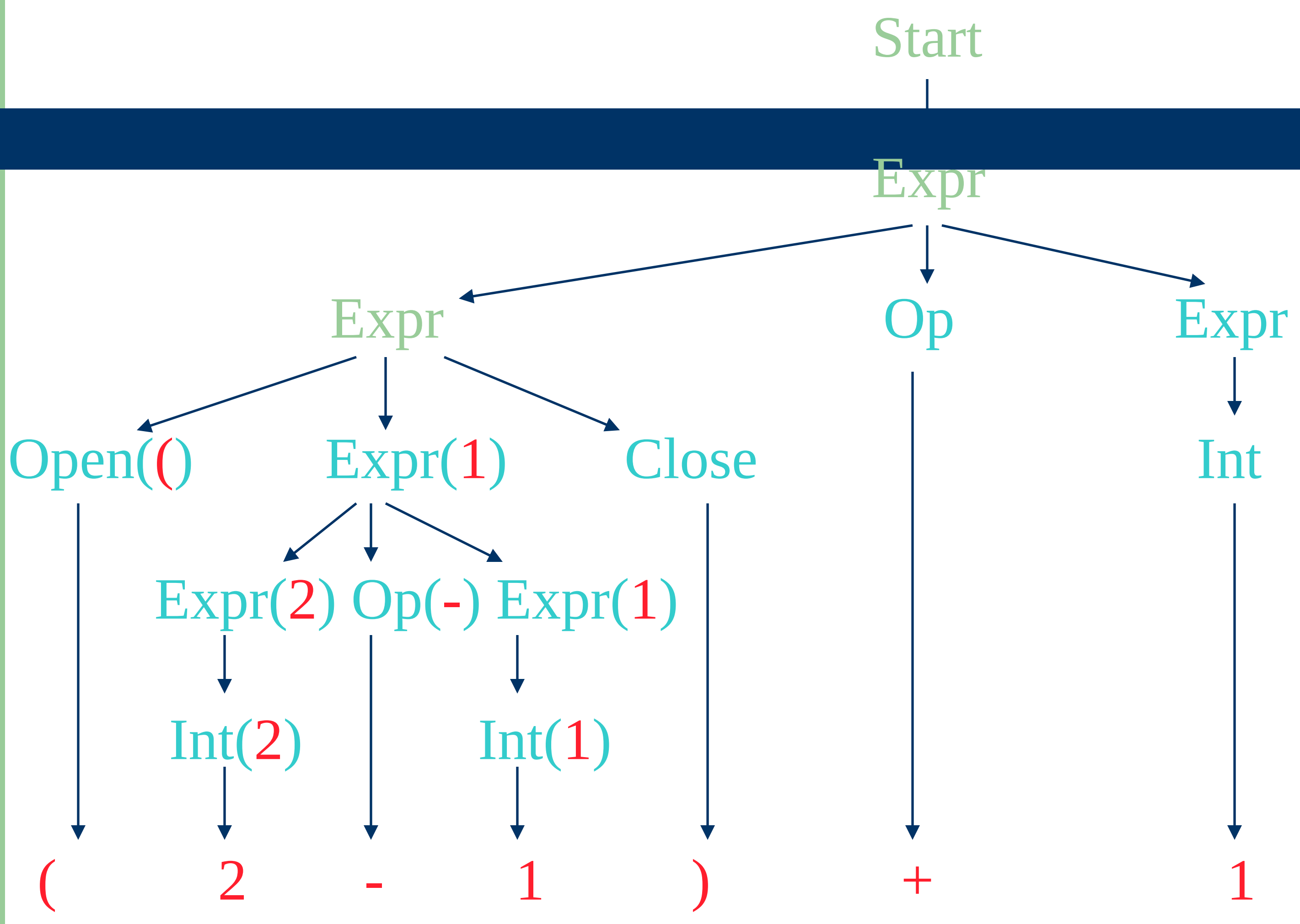
Processing the Tree



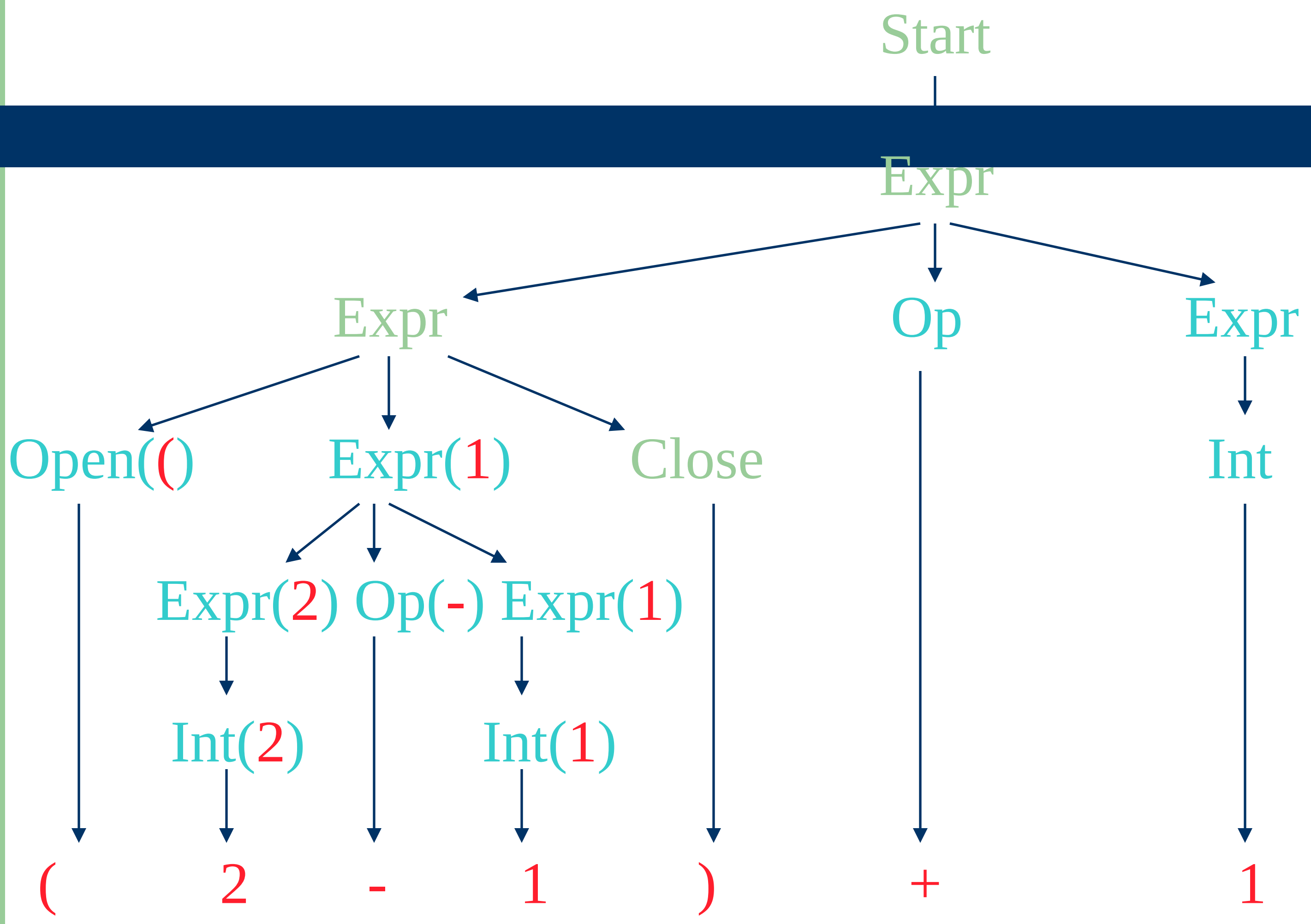
Processing the Tree



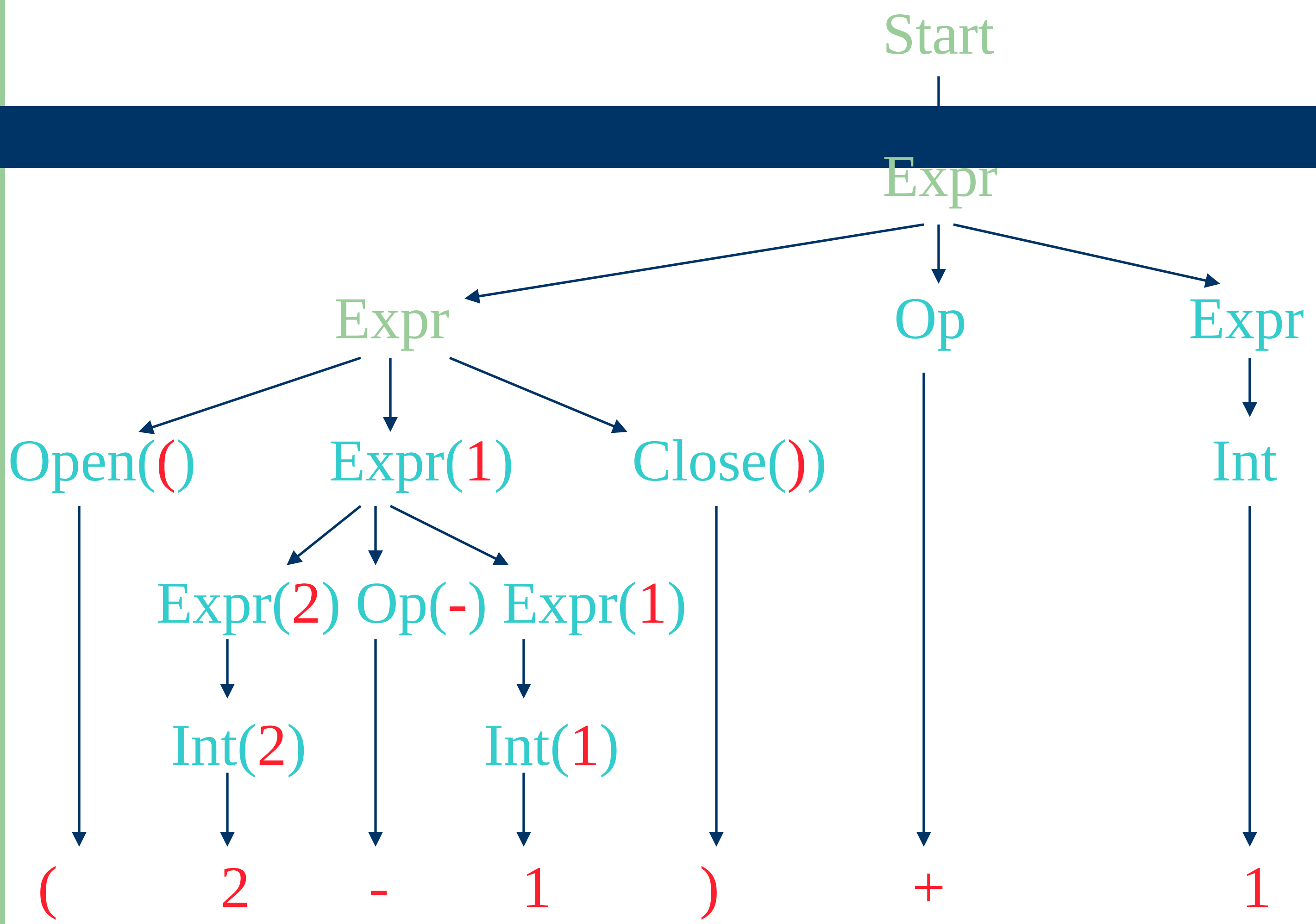
Processing the Tree



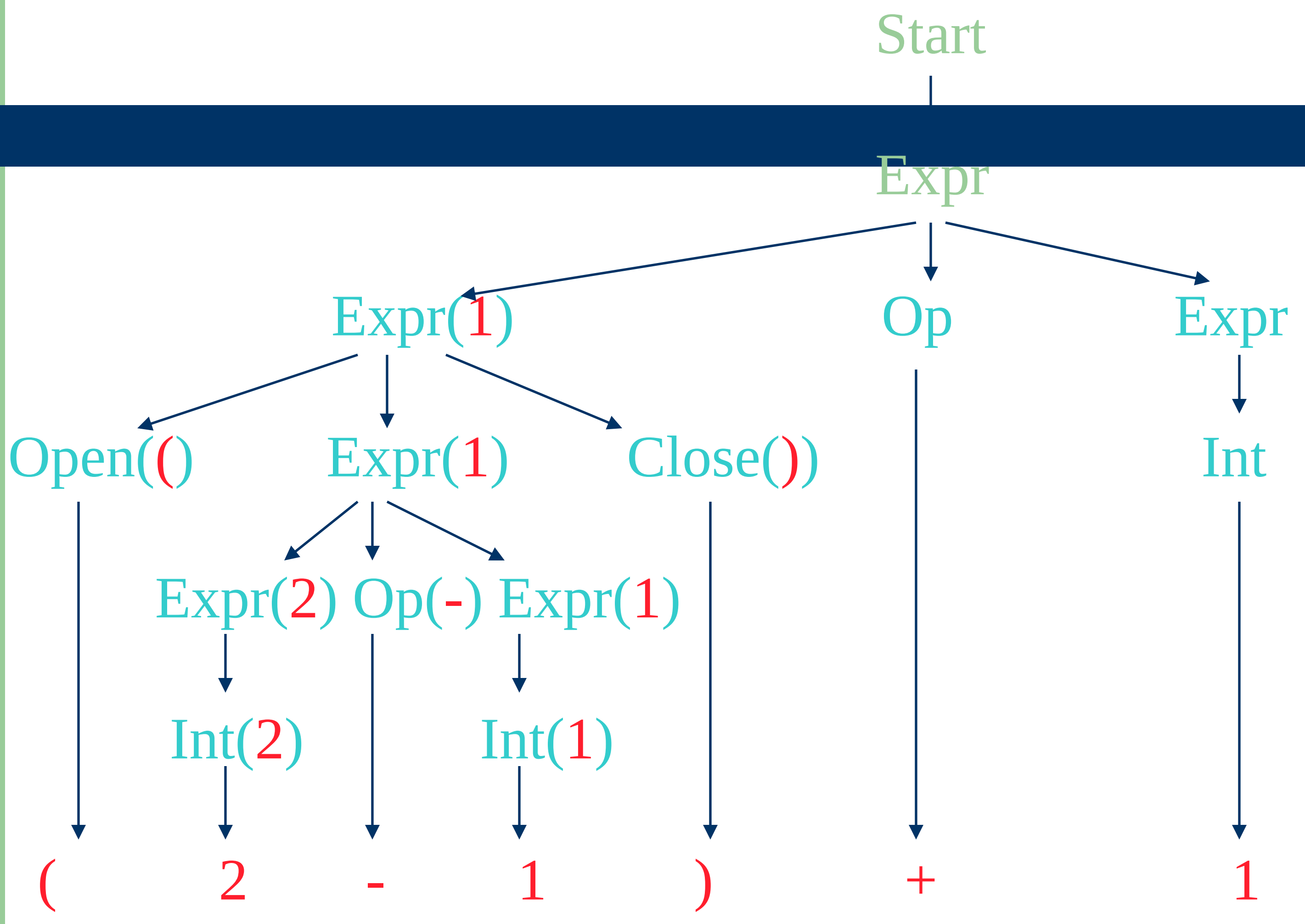
Processing the Tree



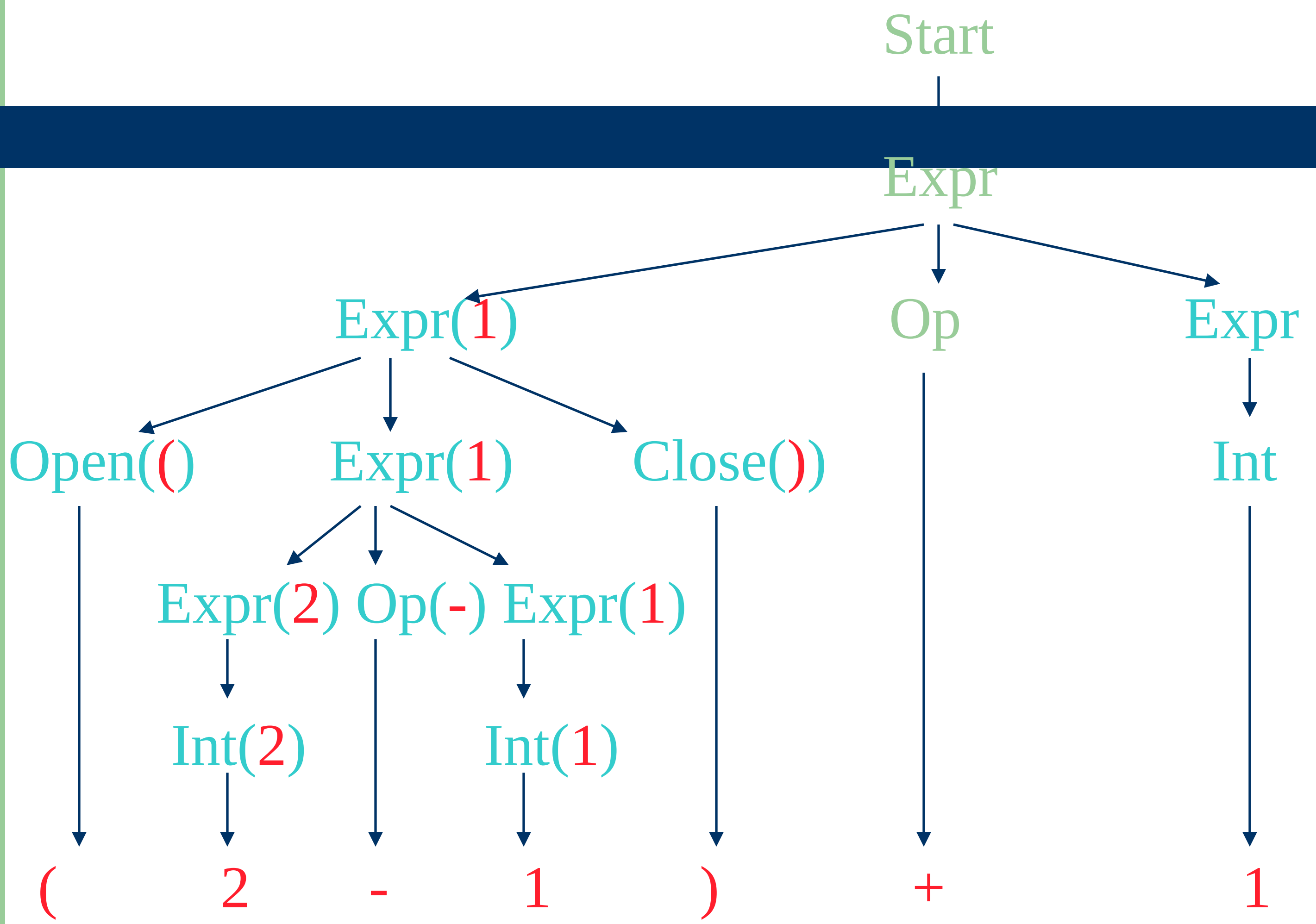
Processing the Tree



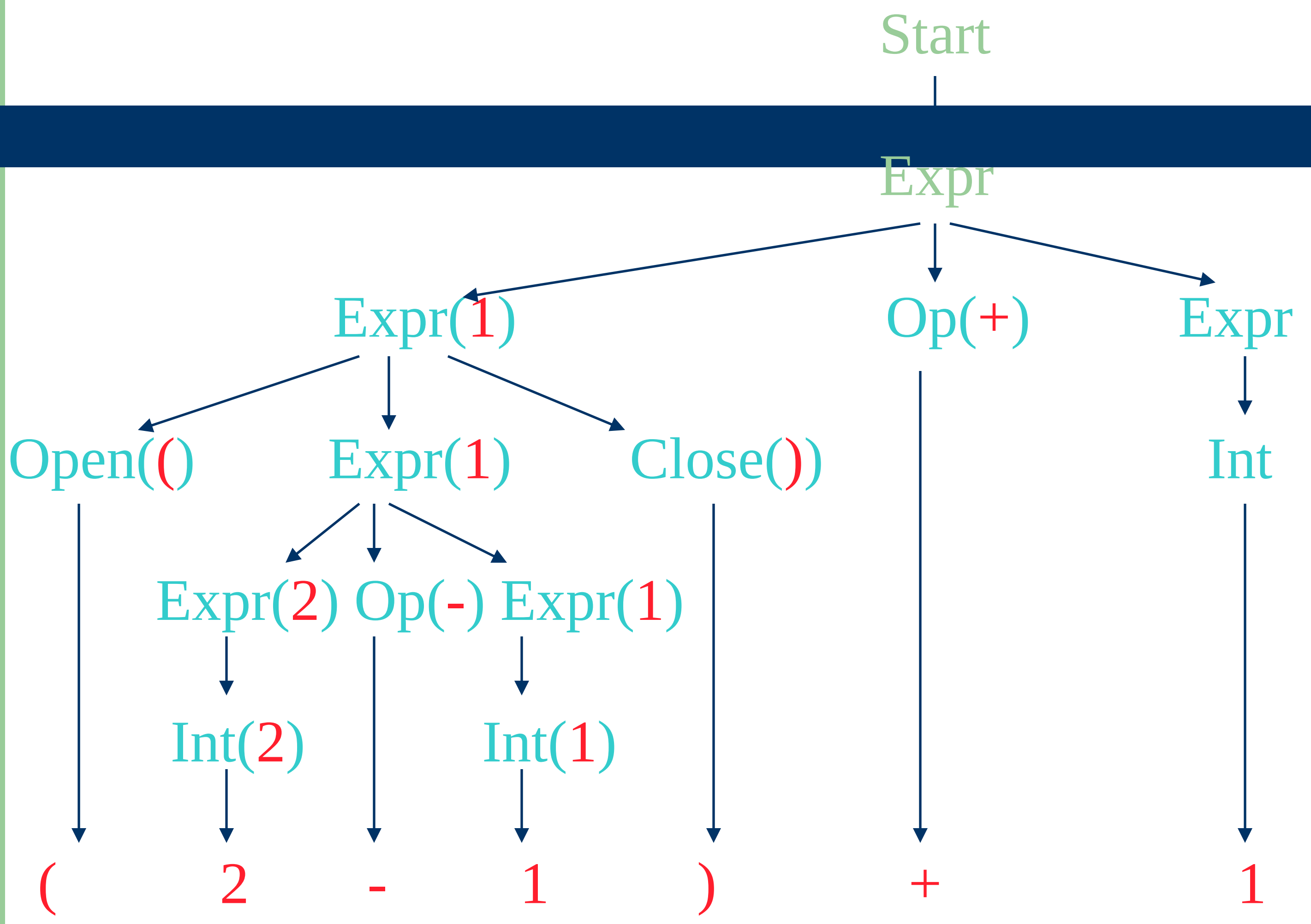
Processing the Tree



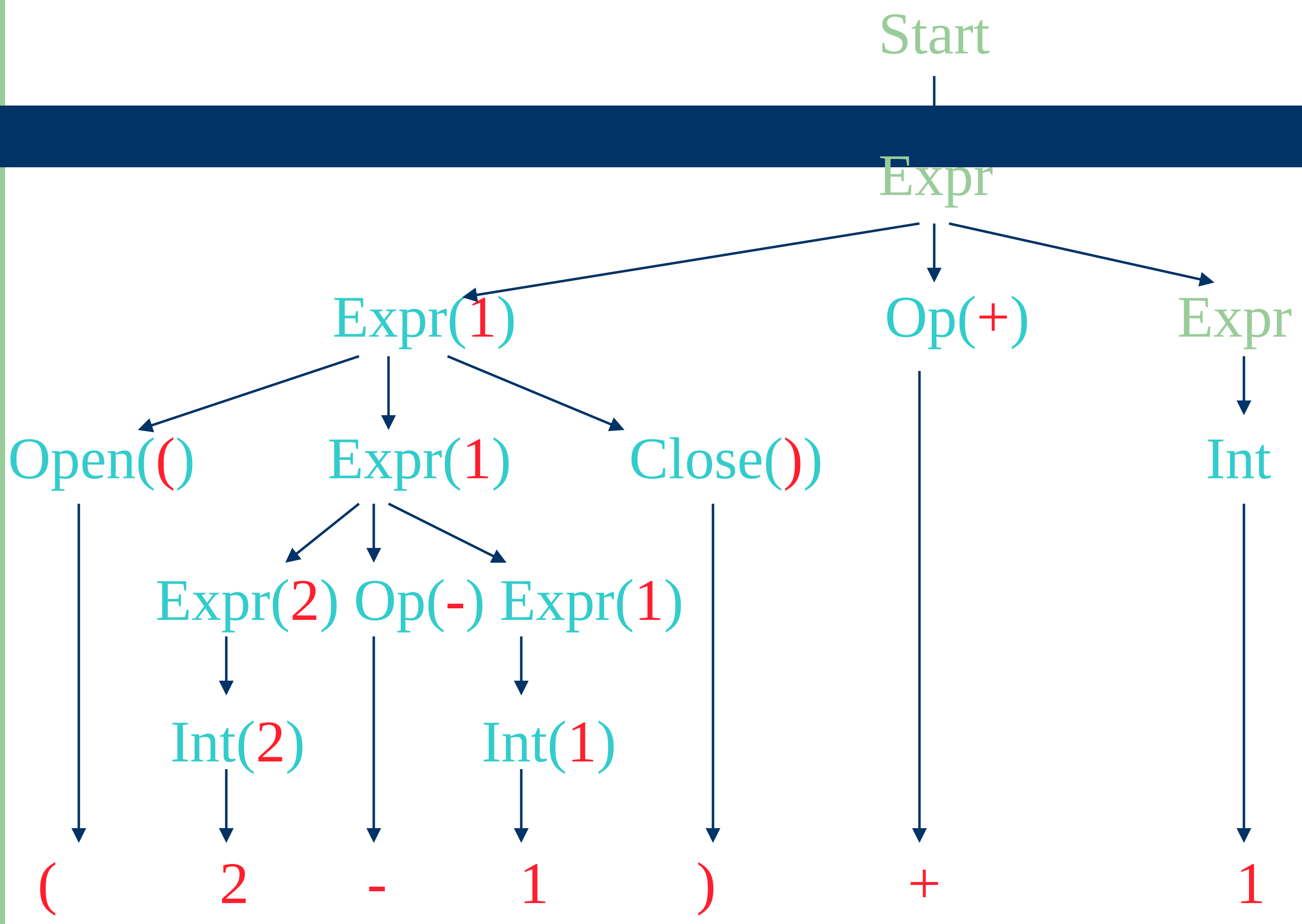
Processing the Tree



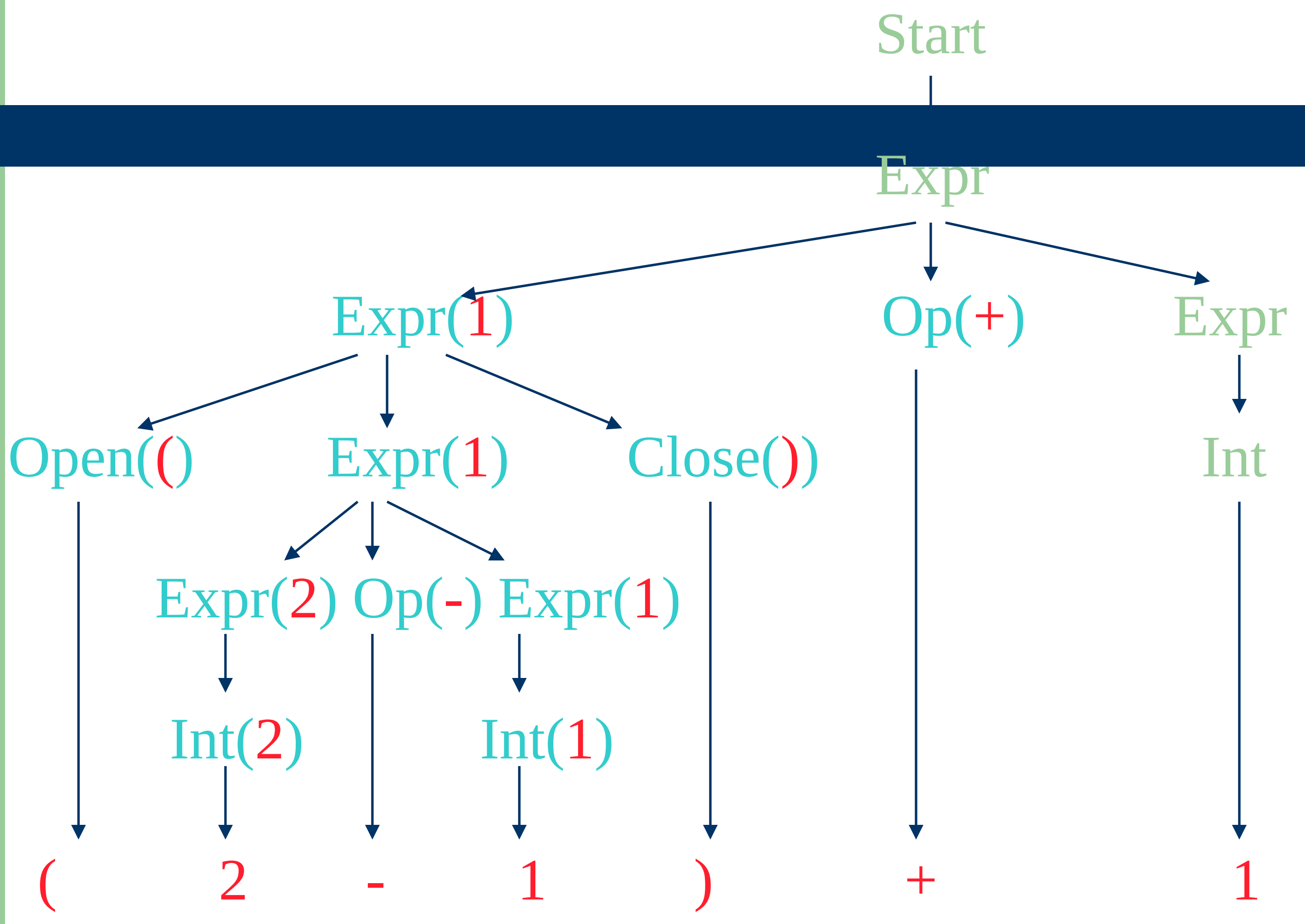
Processing the Tree



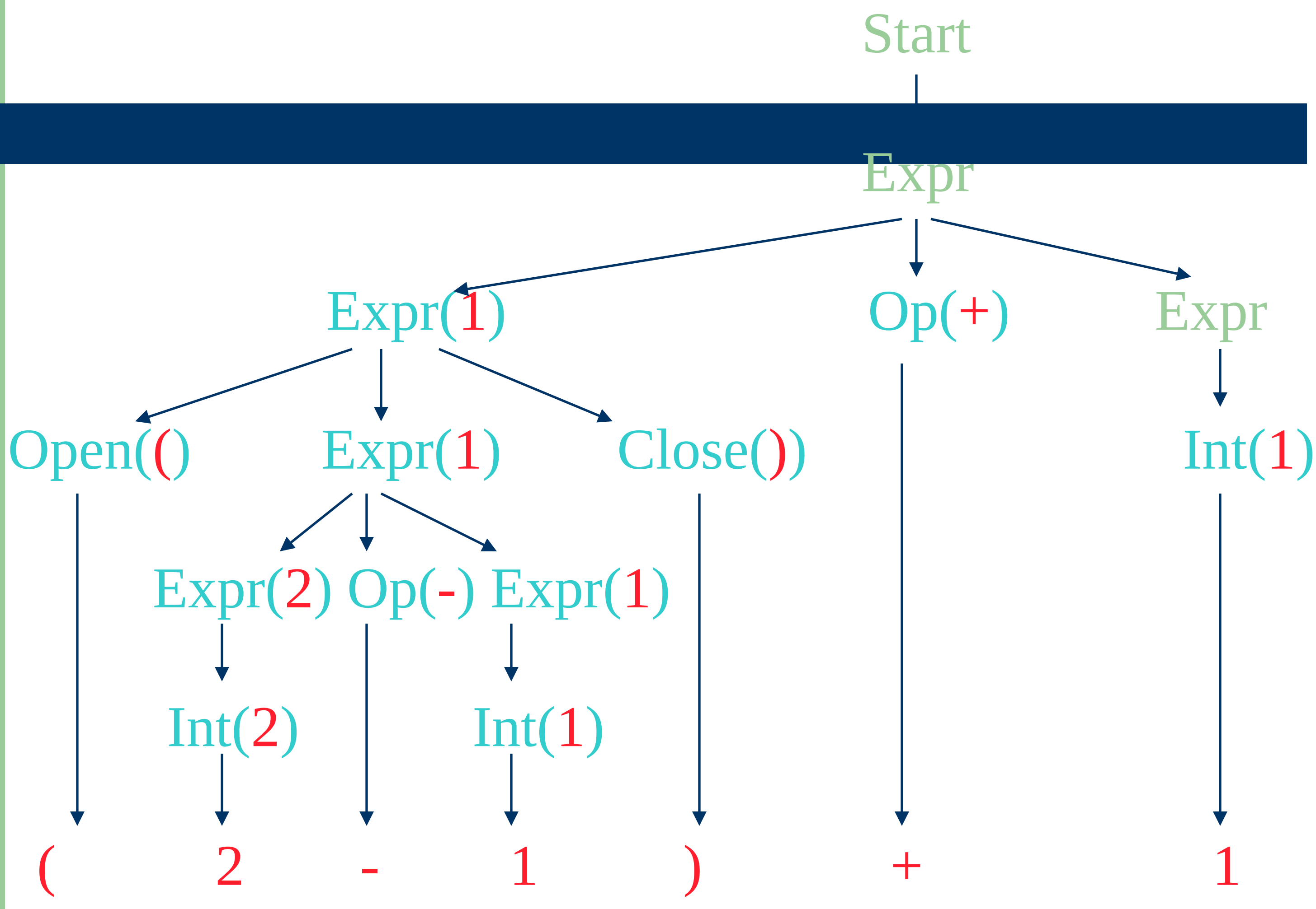
Processing the Tree



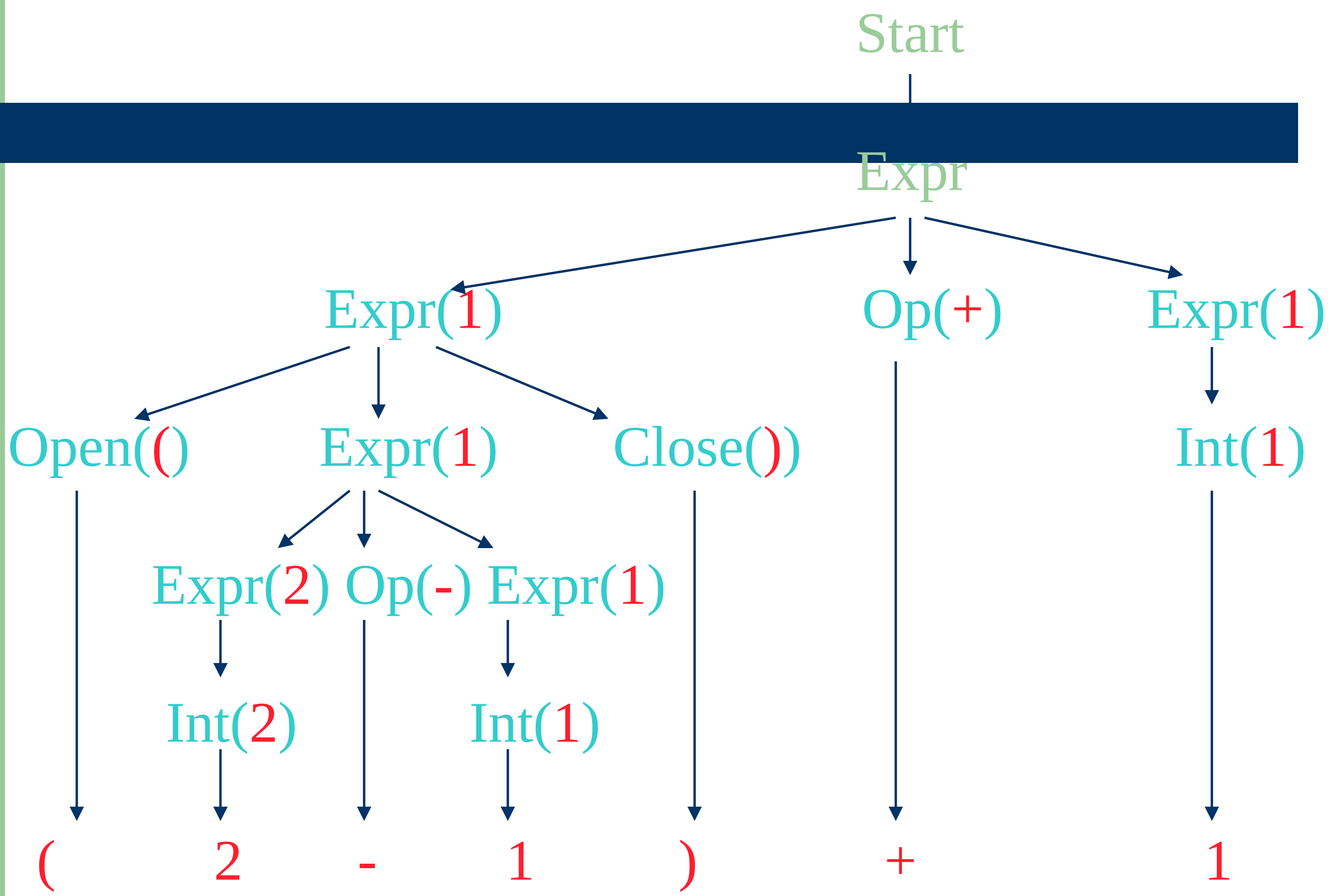
Processing the Tree



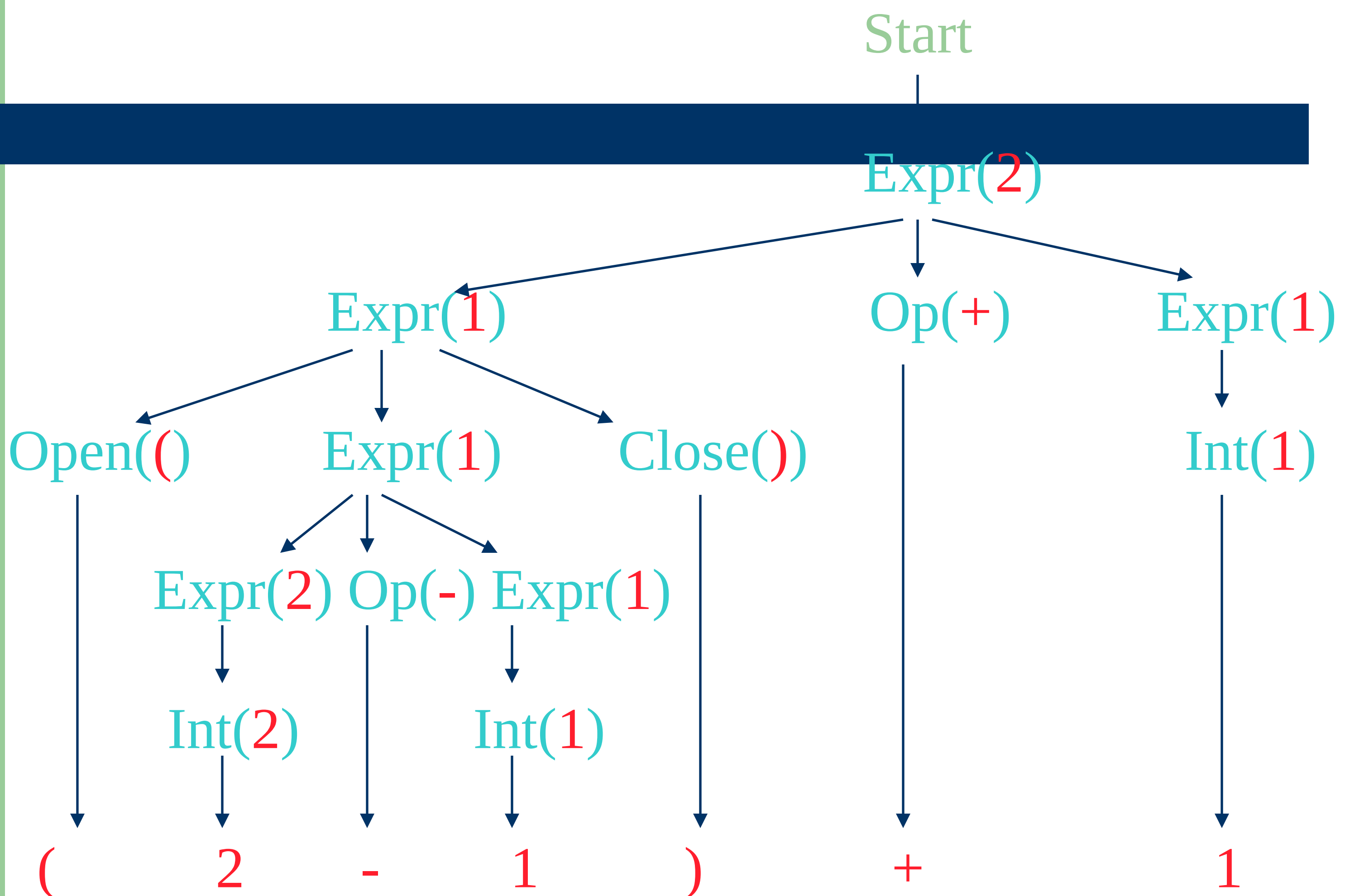
Processing the Tree



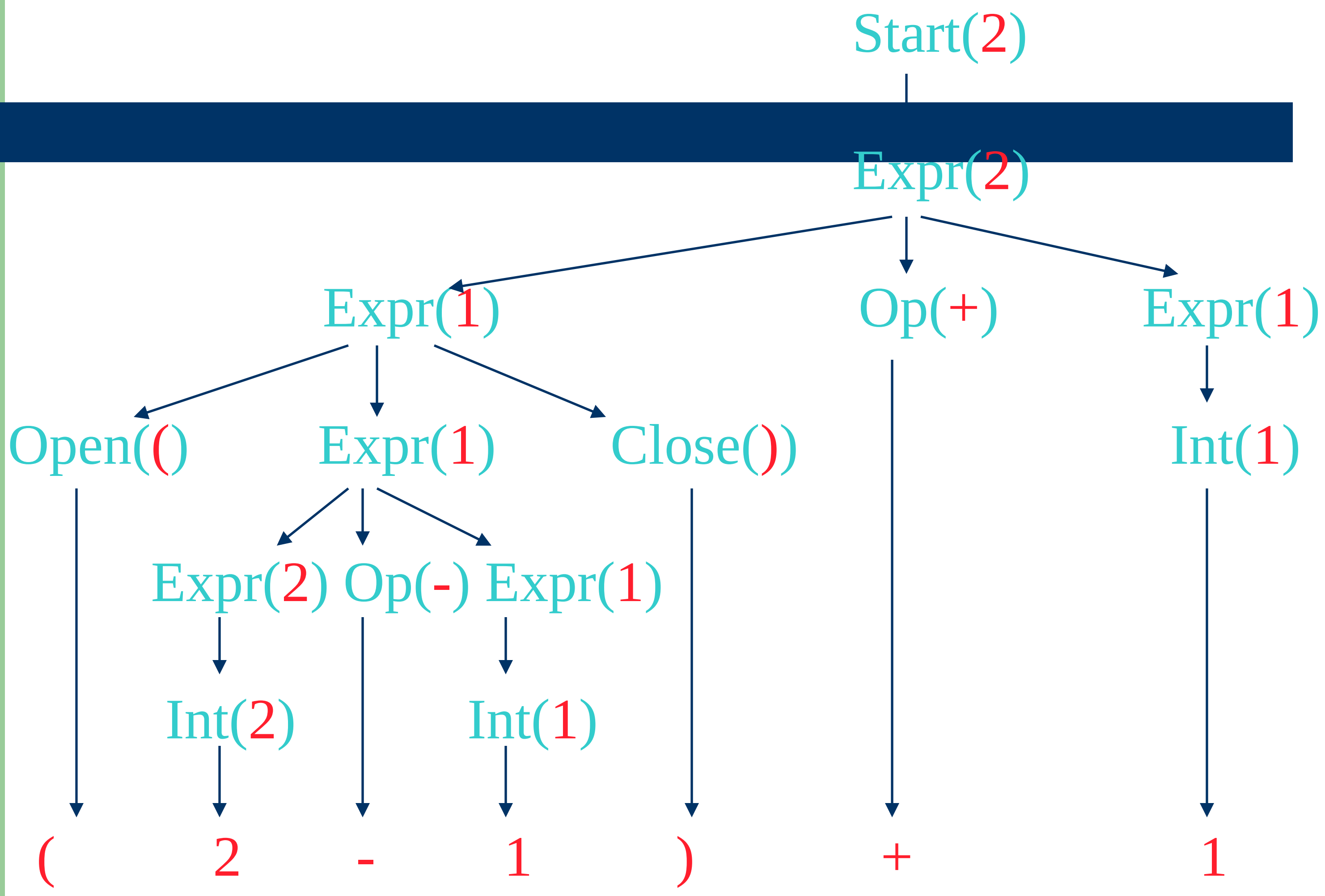
Processing the Tree



Processing the Tree



Processing the Tree



General Grammar

Exp $\longrightarrow$ exp + term

Exp $\longrightarrow$ exp - term

Exp $\longrightarrow$ term

Term $\longrightarrow$ term * Factor

Term $\longrightarrow$ term / Factor

Term $\longrightarrow$ Factor

Factor $\longrightarrow$ digit

Factor $\longrightarrow$ (exp)

CFG - Example

- Grammar for balanced-parentheses language
  - $S \rightarrow (S) S$
  - $S \rightarrow \varepsilon$
    - 1 non-terminal: S
    - 3 terminals: “(”, “)”,ε
    - Start symbol: S
    - 2 productions
- If grammar accepts a string, there is a derivation of that string using the productions
  - How do we produce: (())
  - $S = (S)\ \varepsilon = ((S)\ S)\ \varepsilon = ((\varepsilon)\ \varepsilon\ )\ \varepsilon = (())$

Stack based algorithm

- Push start symbol onto stack
- Replace non-terminal symbol on stack using grammar rules
- Objective is to have something on stack which will match input stream
- If top of stack matches input token, both may be discarded
- If, eventually, both stack and input string are empty then successful parse

Demonstration

- Grammar

$$S \rightarrow (\; S \;) \; S \mid \varepsilon$$

- Generates strings of balanced parentheses
- S
- (S) S
- ((S) S) S
- ((S) S) (S) S
- (( )) ( )

Demonstration

The Grammar

$S \rightarrow (S) S \mid \varepsilon$

The Input

`()$`

- We mark the bottom of the stack with a dollar sign.
- Note also that the input is terminated with a dollar sign representing end of input

Demonstration

The Grammar

$S \rightarrow (S) S \mid \varepsilon$

The Input

`()$`

- Start by pushing the start symbol onto the stack

Demonstration

The Grammar

`S → ( S ) S | ε`

The Input

`()$`

- Replace it with a rule from the grammar: `S → ( S ) S`
- Note that the rule is pushed onto the stack from right to left

(
S
)
S
$

Demonstration

```
The Grammar
S → ( S ) S | ε
```

```
The Input
()$
```

- Now we match the top of the stack with the next input character

(
S
)
S
$

Demonstration

The Grammar

$S \rightarrow (S) S \mid \varepsilon$

The Input

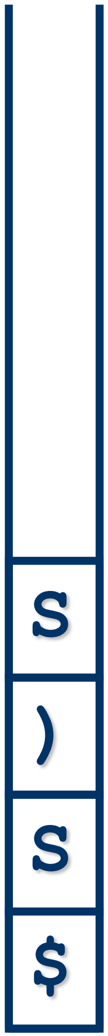
`()$`

- Characters matched are removed from both stack and input stream

Demonstration

The Grammar
S → (S) S | ε

The Input
)$



- Characters matched are removed from both stack and input stream

Demonstration

The Grammar

`S → ( S ) S | ε`

The Input

`)$`

- Now we use the rule: `S → ε`

S
)
S
$

Demonstration

The Grammar

`S → ( S ) S | ε`

The Input

`)$`

- Now we use the rule: `S → ε`

Demonstration

The Grammar

`S → ( S ) S | ε`

The Input

`)$`

- We can again match

Demonstration

The Grammar

`S → ( S ) S | ε`

The Input

`$`

- and remove matches

Demonstration

The Grammar

`S → ( S ) S | ε`

The Input

`$`

- One more application of the rule: $S \rightarrow \varepsilon$

S

$

Demonstration

The Grammar

$S \to (S) S \mid \varepsilon$

The Input

$

- One more application of the rule: $S \to \varepsilon$

$

Demonstration

The Grammar

S → (S) S | ε

The Input

$

- Now finding both stack and input are at $ we conclude successful parse

$

THANKS