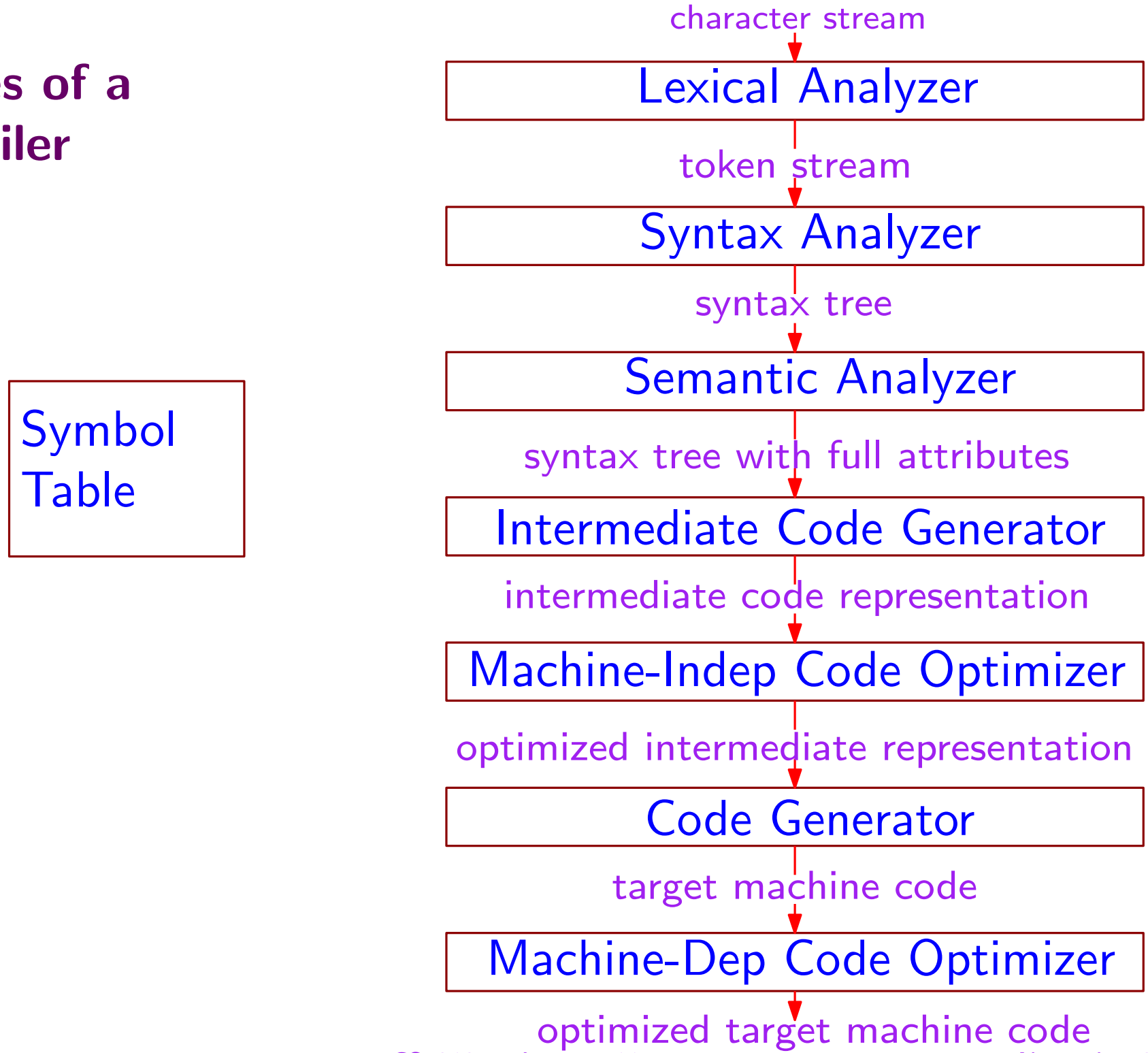
Syntactic Analysis

Phases of a Compiler

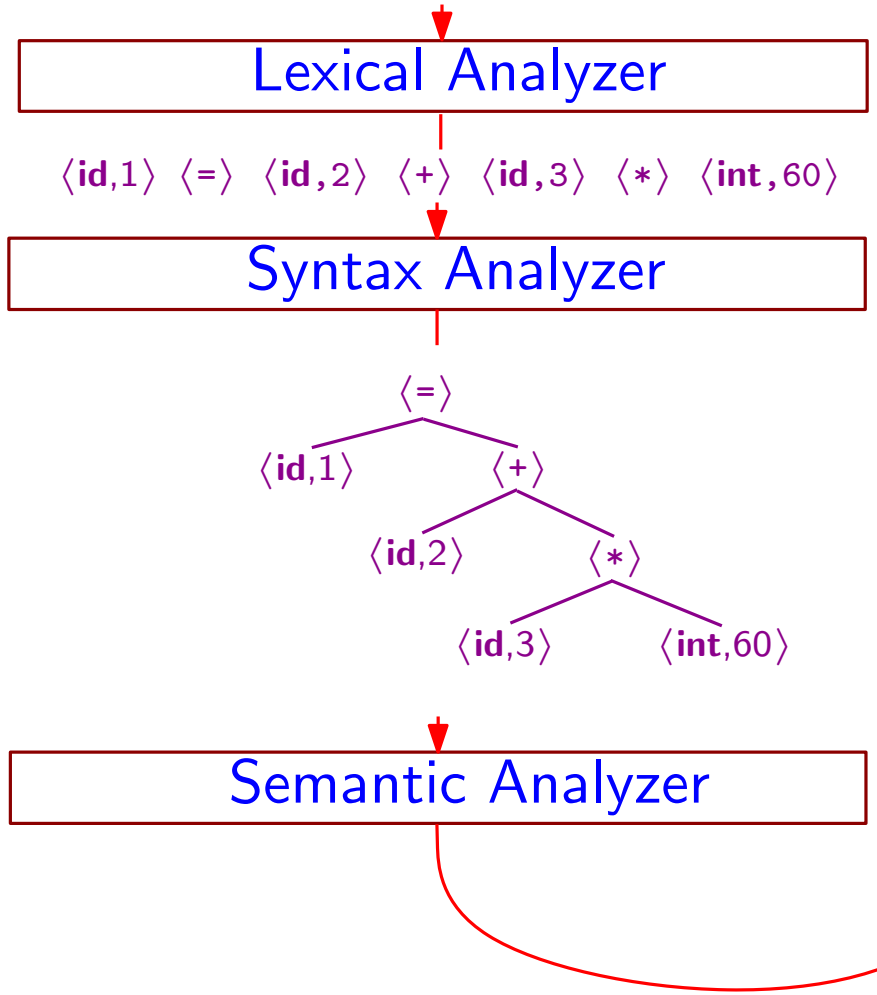


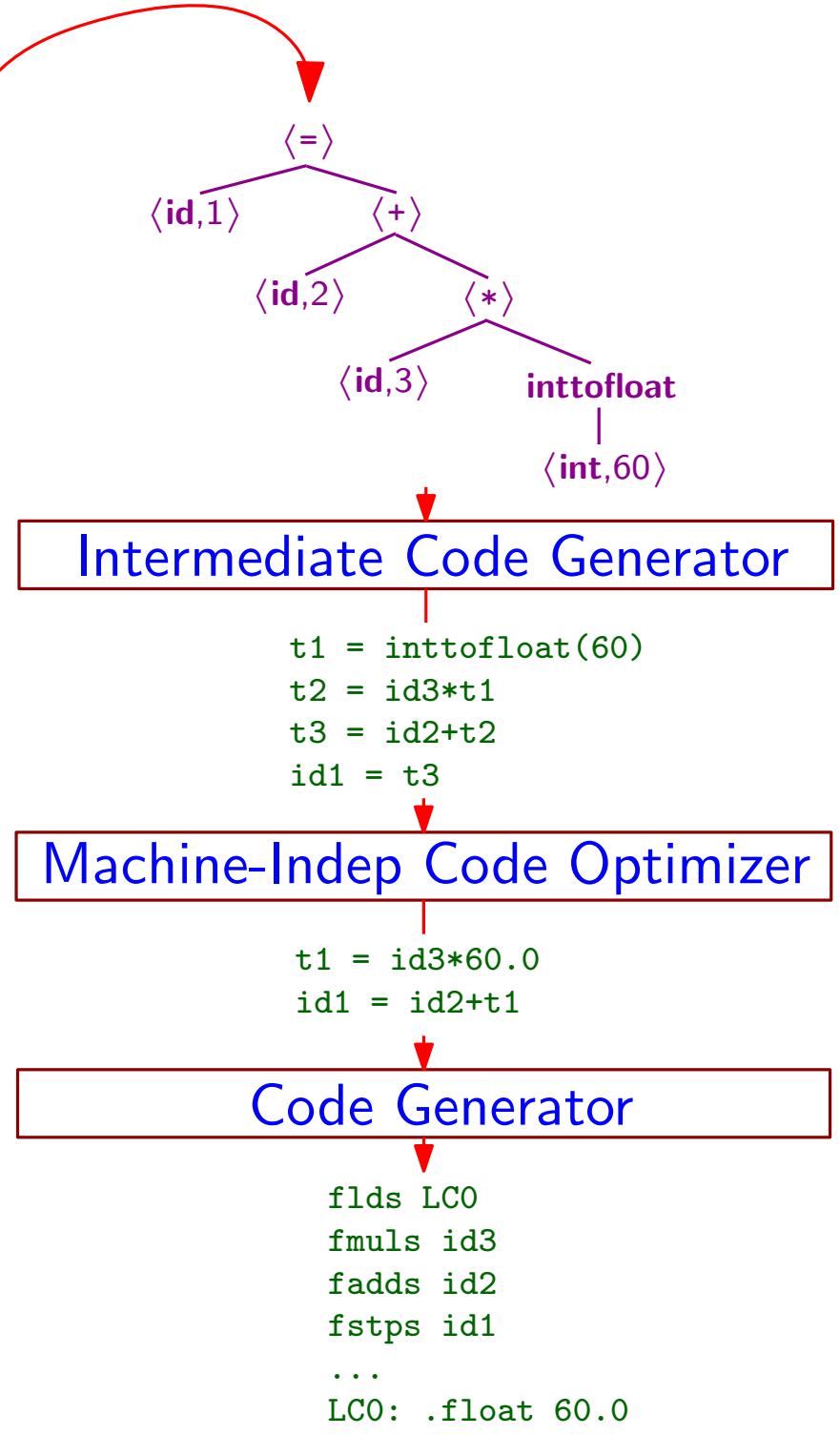
Example

Symbol Table

1	position	···
2	initial	···
3	rate	···

```
position = initial + rate*60
```





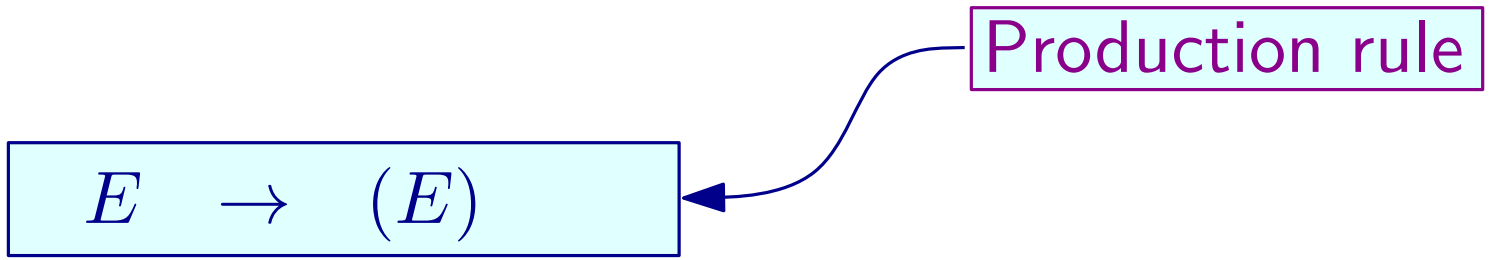
Grammars

- Mathematical concept in the area of *formal languages*
- Language generator: specifies a mechanism for generating all elements of the language.
  - The language is the set of strings that can be generated.
  - Can be converted into an *acceptor*: a procedure that tests whether a string is in the language or not.
- Practitioner's approach: introduce concepts step-by-step

Production Rules

$$
\begin{aligned}
E &\rightarrow (E) \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow D \\
D &\rightarrow 0D \\
D &\rightarrow 1D \\
D &\rightarrow 0 \\
D &\rightarrow 1
\end{aligned}
$$

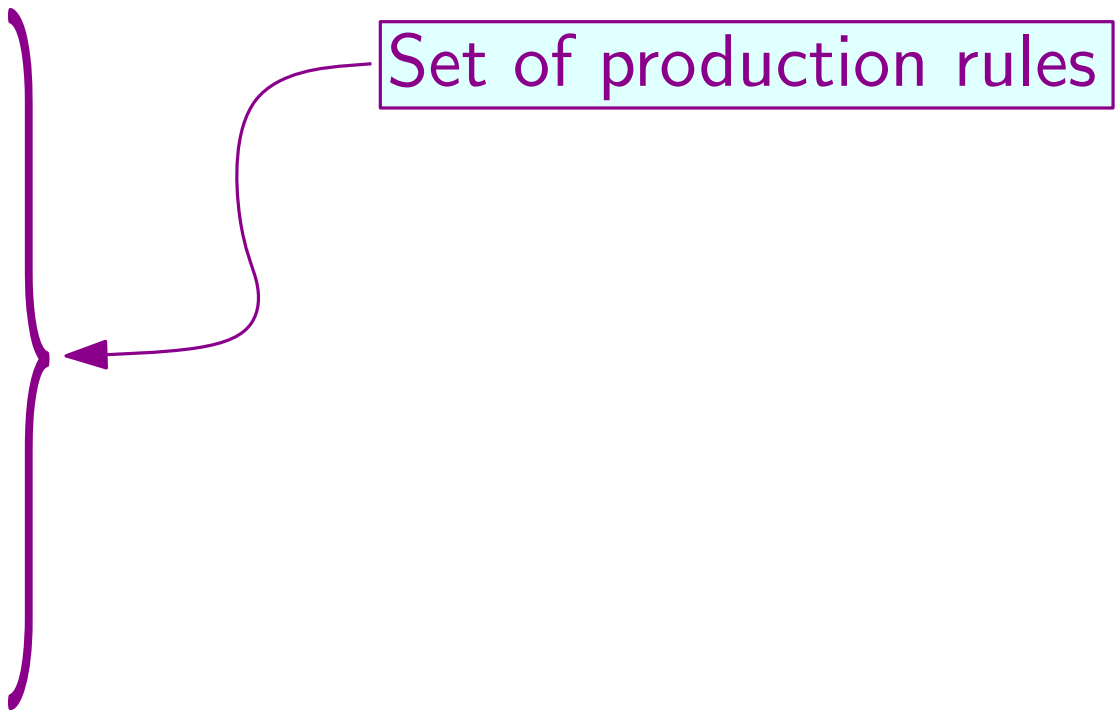
Production Rules



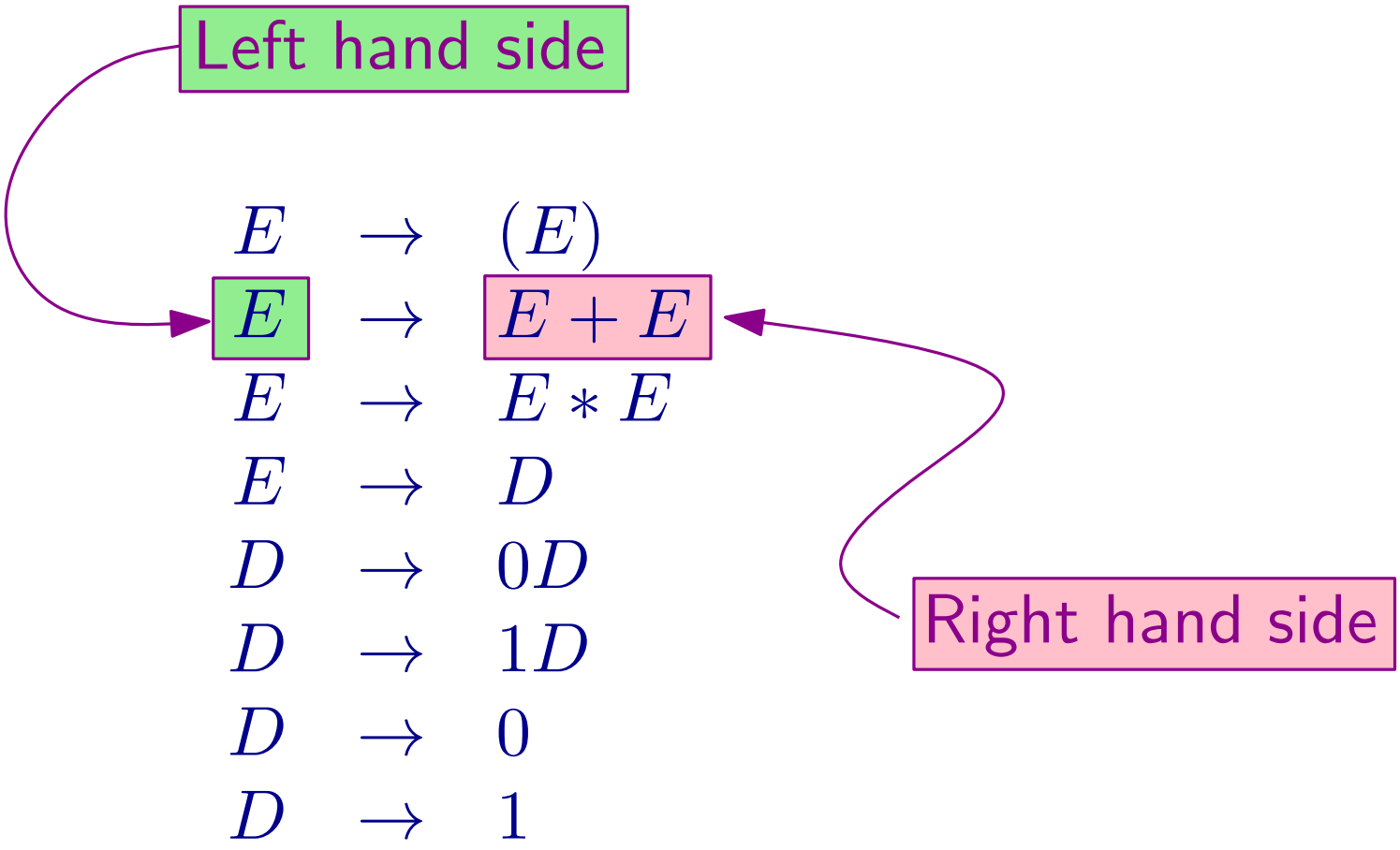
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

Production Rules

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



Production Rules



Left hand side can be *rewritten* as the right hand side.

Production Rules

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

Non-terminals

- Denoted by capitals
- Not part of the generated language.
- An aid to generating the language.

Production Rules

$$
\begin{aligned}
E &\rightarrow (E) \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow D \\
D &\rightarrow 0D \\
D &\rightarrow 1D \\
D &\rightarrow 0 \\
D &\rightarrow 1
\end{aligned}
$$

Terminals:

- The rest of the symbols.
- Part of the generated language.

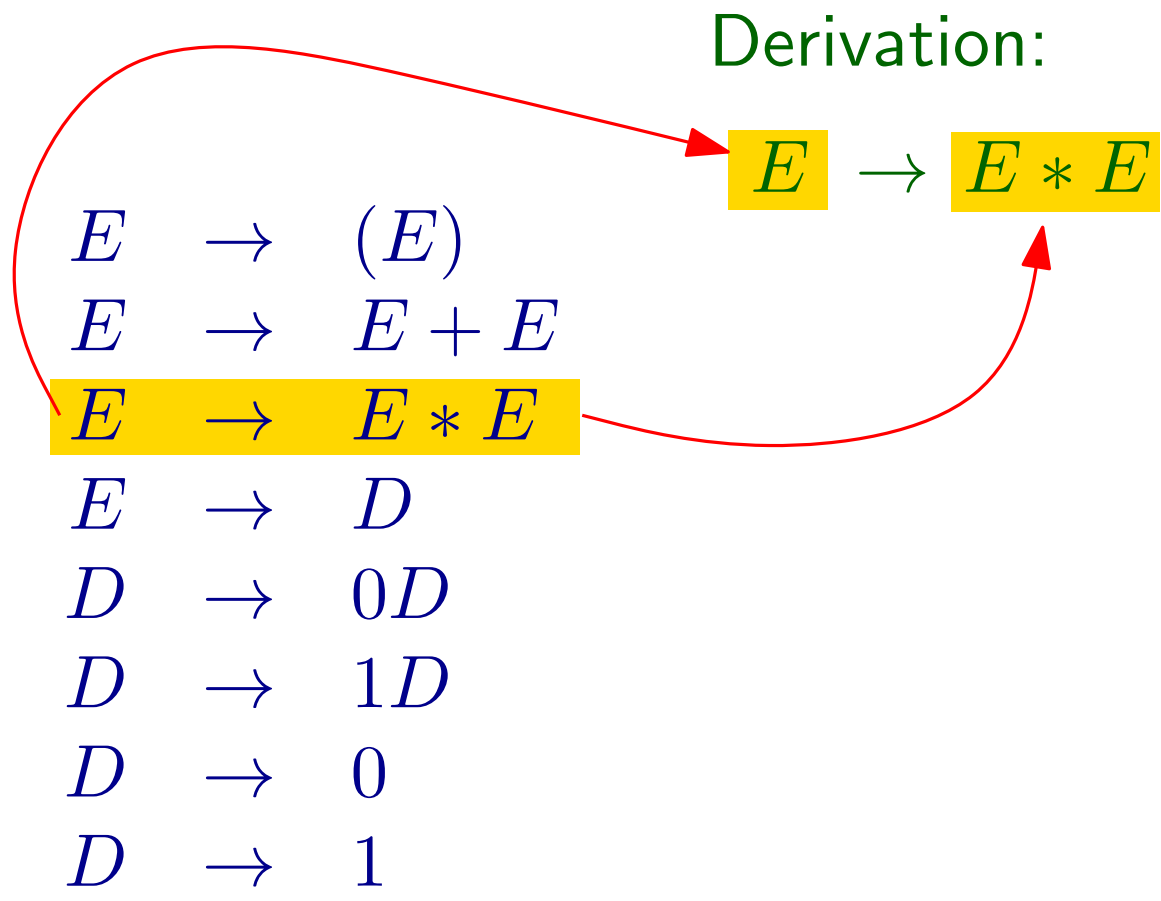
Production Rules

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

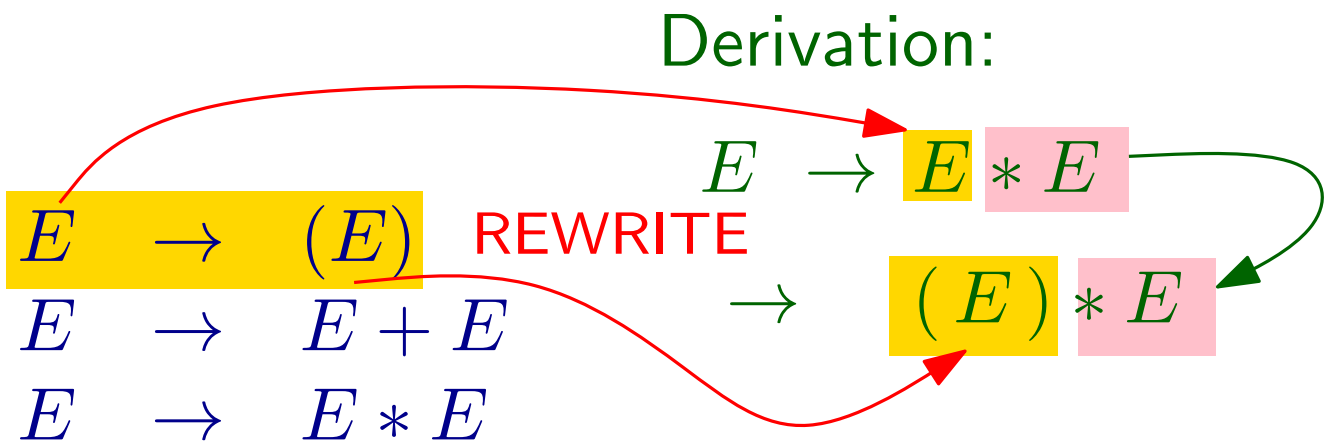
Derivation:

$$E \rightarrow$$

Production Rules

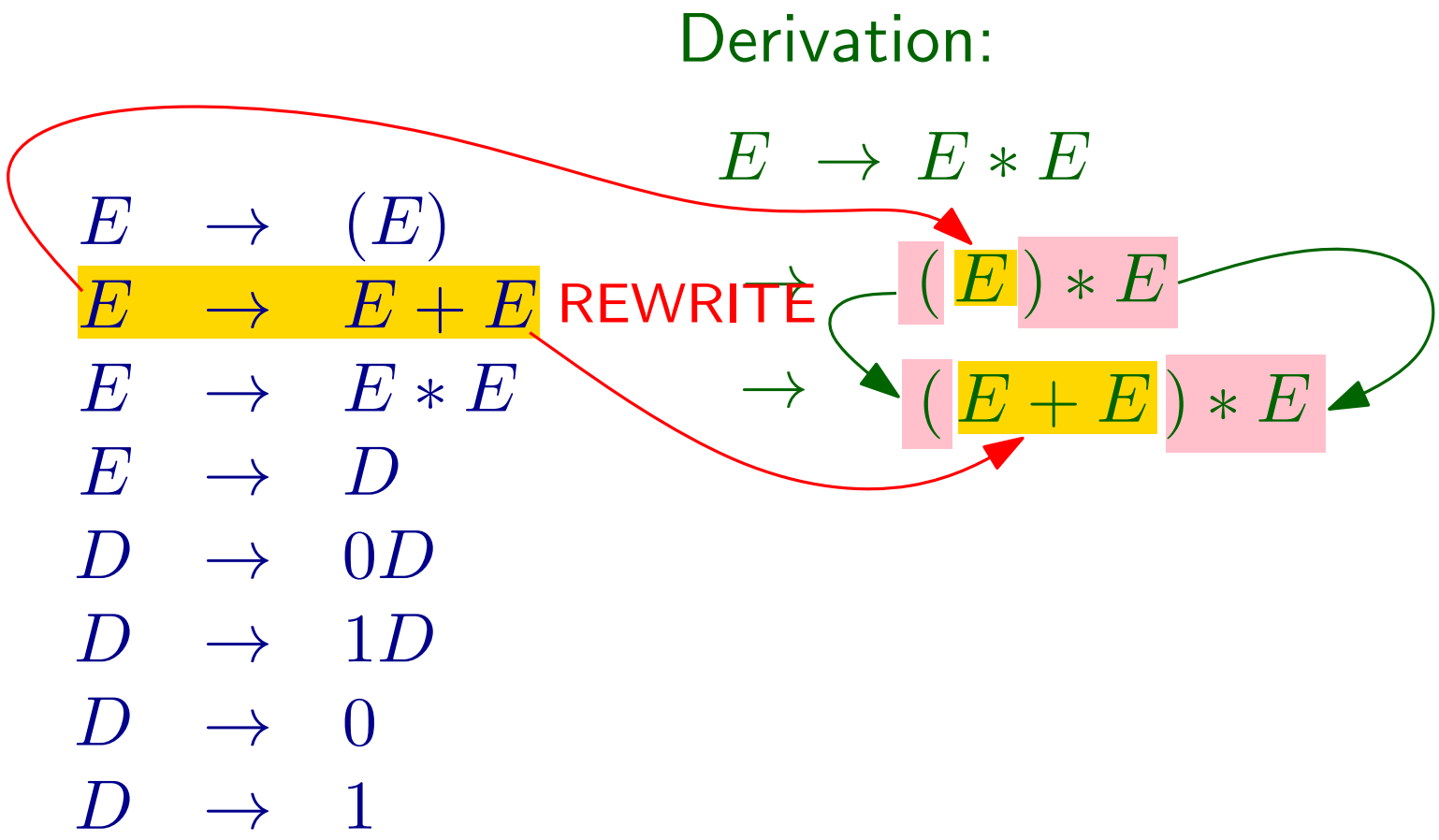


Production Rules

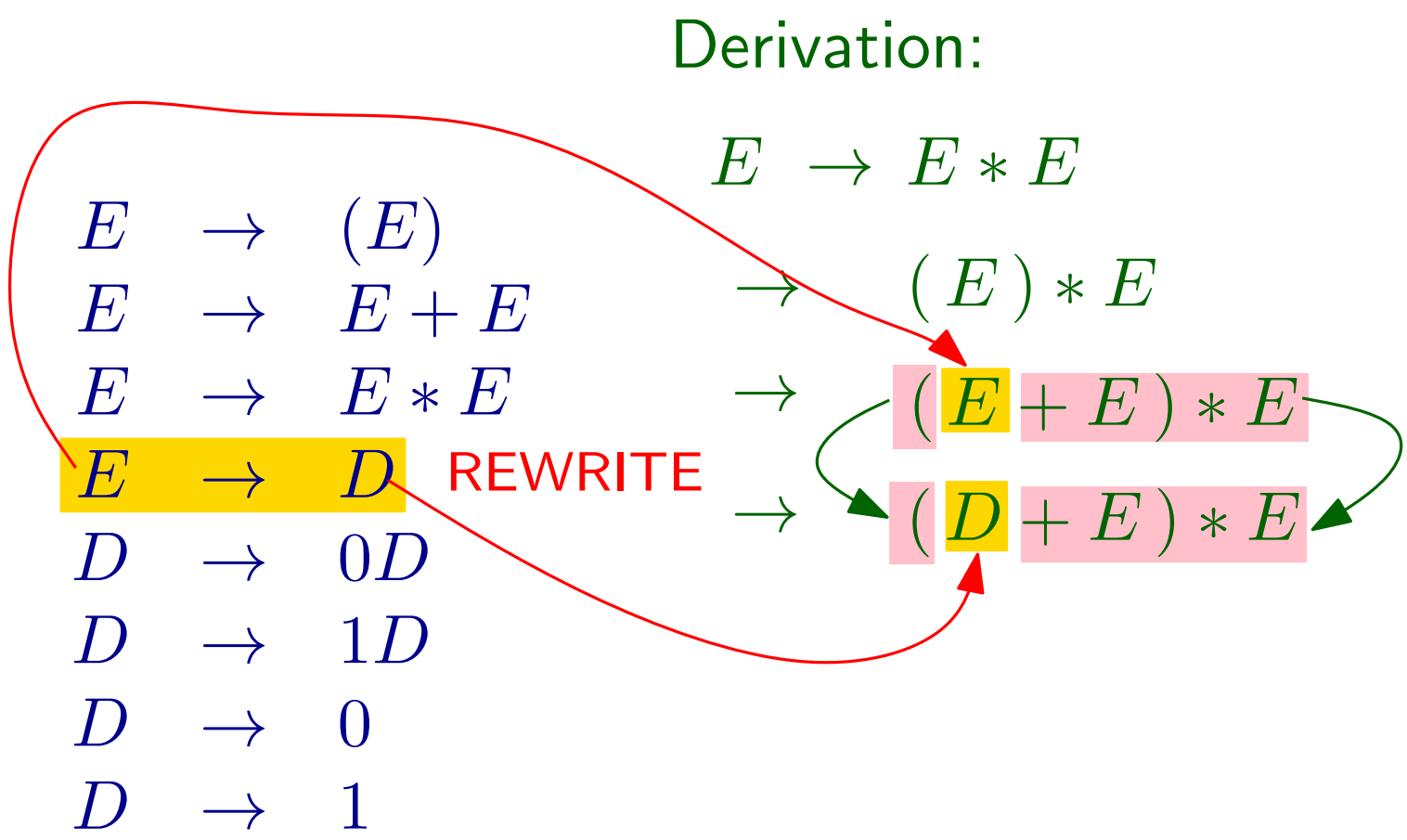


$$
\begin{array}{lcl}
E & \to & (E) \\
E & \to & E + E \\
E & \to & E * E \\
E & \to & D \\
D & \to & 0D \\
D & \to & 1D \\
D & \to & 0 \\
D & \to & 1
\end{array}
$$

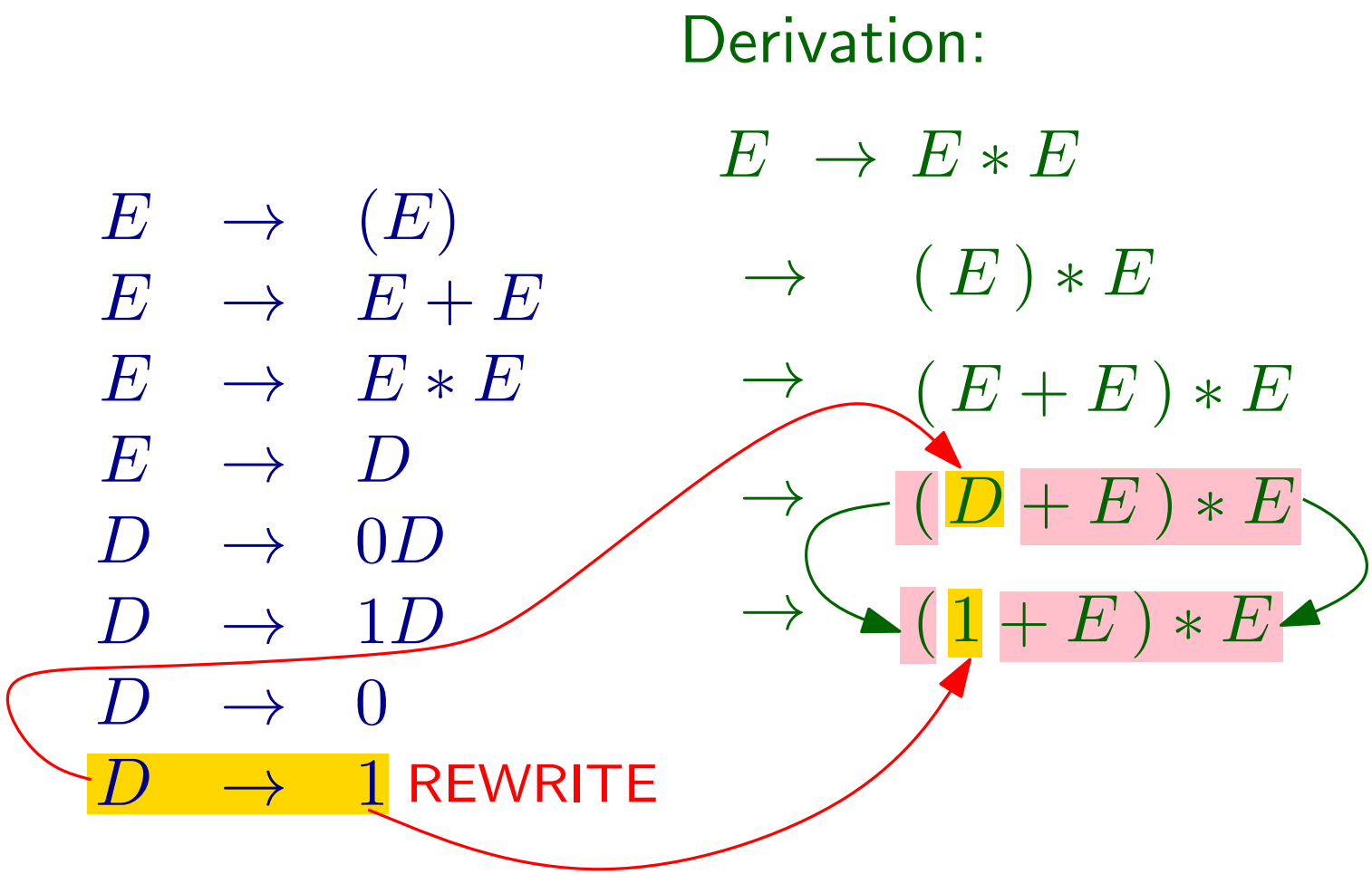
Production Rules



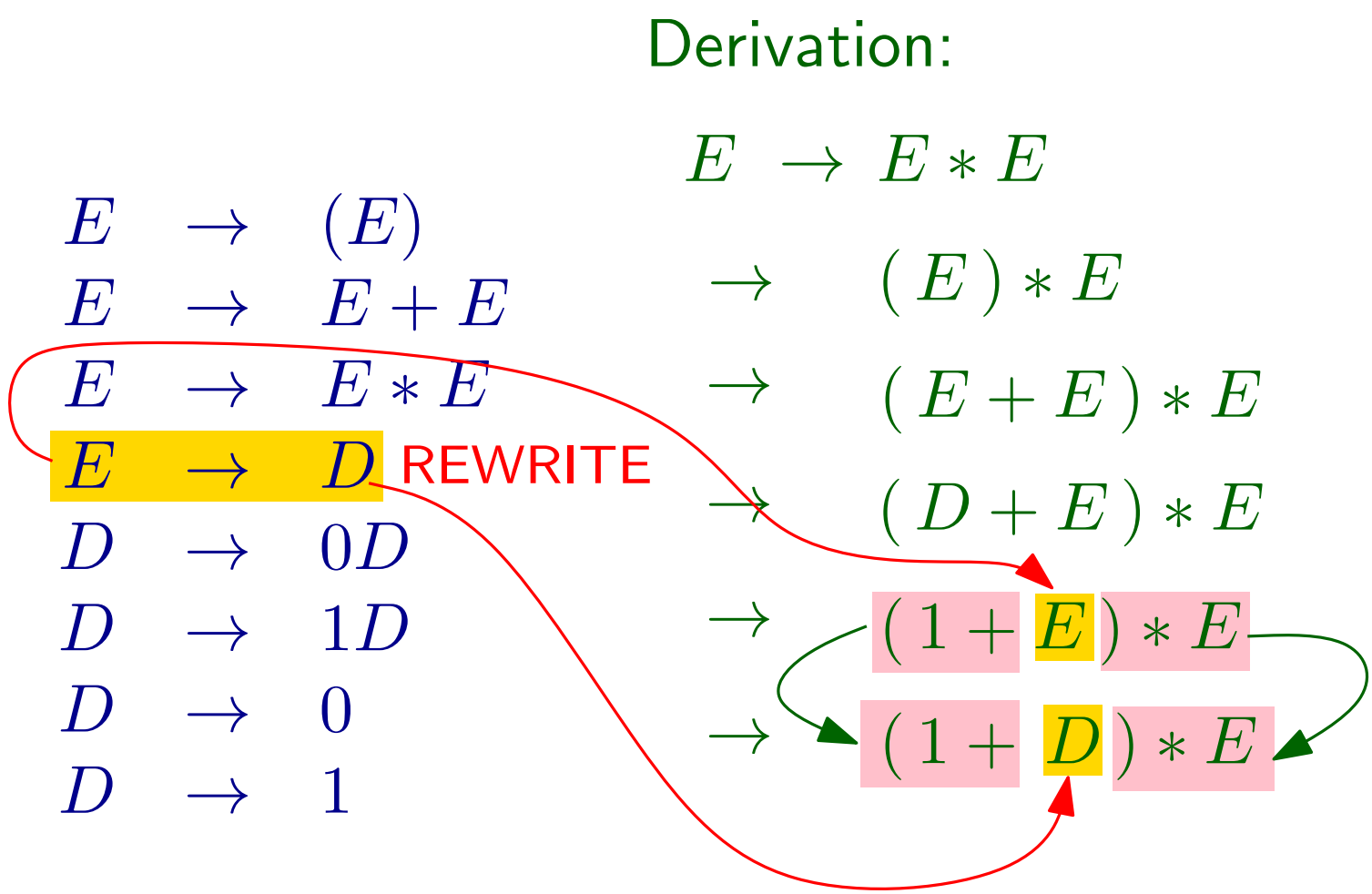
Production Rules



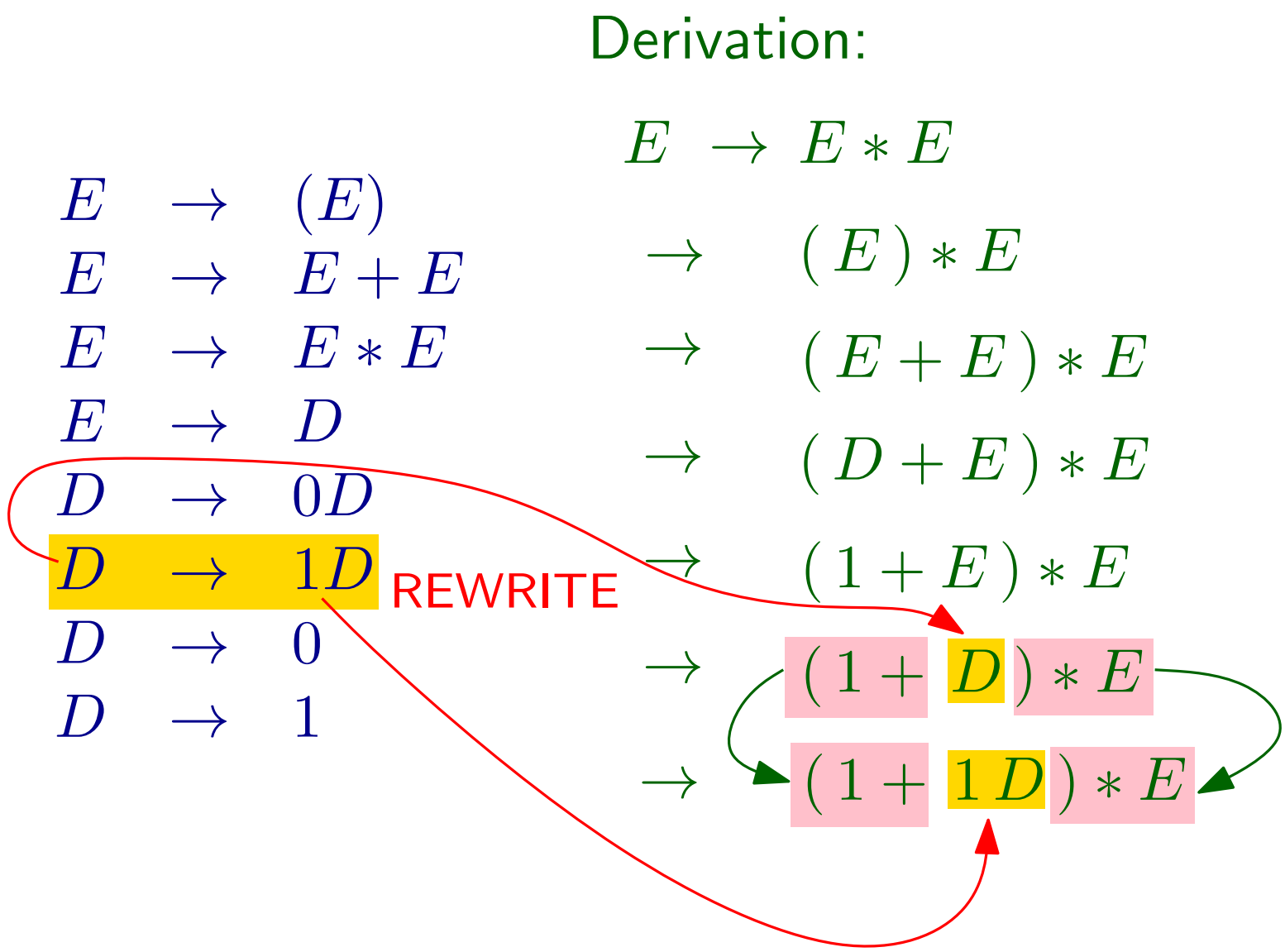
Production Rules



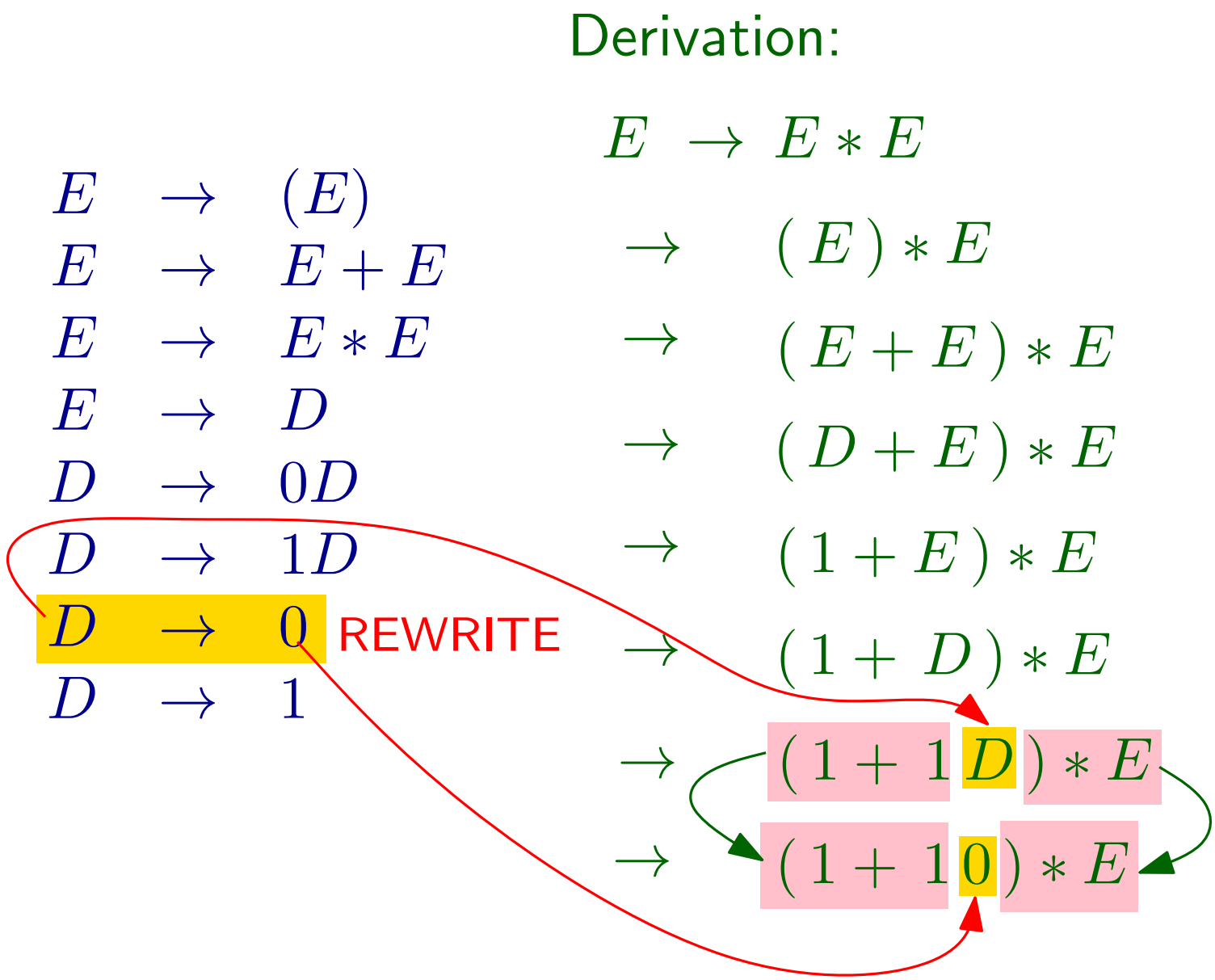
Production Rules



Production Rules



Production Rules



Production Rules

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

Derivation:

$$
\begin{array}{lcl}
E & \rightarrow & E * E \\
 & \rightarrow & (E) * E \\
 & \rightarrow & (E + E) * E \\
 & \rightarrow & (D + E) * E \\
 & \rightarrow & (1 + E) * E \\
 & \rightarrow & (1 + D) * E \\
 & \rightarrow & (1 + 1D) * E \\
 & \rightarrow & (1 + 10) * E \\
 & \rightarrow & (1 + 10) * D \\
 & \rightarrow & (1 + 10) * 1
\end{array}
$$

Production Rules

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

Derivation:

$$(1 + 10) * 1 \in \mathcal{L}(E)$$

$$
\begin{array}{lcl}
 & \rightarrow & (1 + E) * E \\
 & \rightarrow & (1 + D) * E \\
 & \rightarrow & (1 + 1D) * E \\
 & \rightarrow & (1 + 10) * E \\
 & \rightarrow & (1 + 10) * D \\
 & \rightarrow & (1 + 10) * 1
\end{array}
$$

Production Rules

Language generated by non-terminal E

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

Derivation:

$$\underbrace{(1 + 10) * 1}_{\text{terminals only}} \in \mathcal{L}(E)$$

Similarly:

$$1001 \in \mathcal{L}(D)$$

$$
\begin{array}{lcl}
 & \rightarrow & (1 + 10) * E \\
 & \rightarrow & (1 + 10) * D \\
 & \rightarrow & (1 + 10) * 1
\end{array}
$$

Formal Definition of Grammar

Grammar: tuple $G \equiv \langle \Sigma, N, \Pi, S \rangle$

Σ - alphabet of *terminal symbols*

N - set of *non-terminal symbols*

Π - set of production rules

S - *start non-terminal*, $S \in N$

$\mathcal{L}(G) \equiv \mathcal{L}(S)$ (contains only strings of terminal symbols)

In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

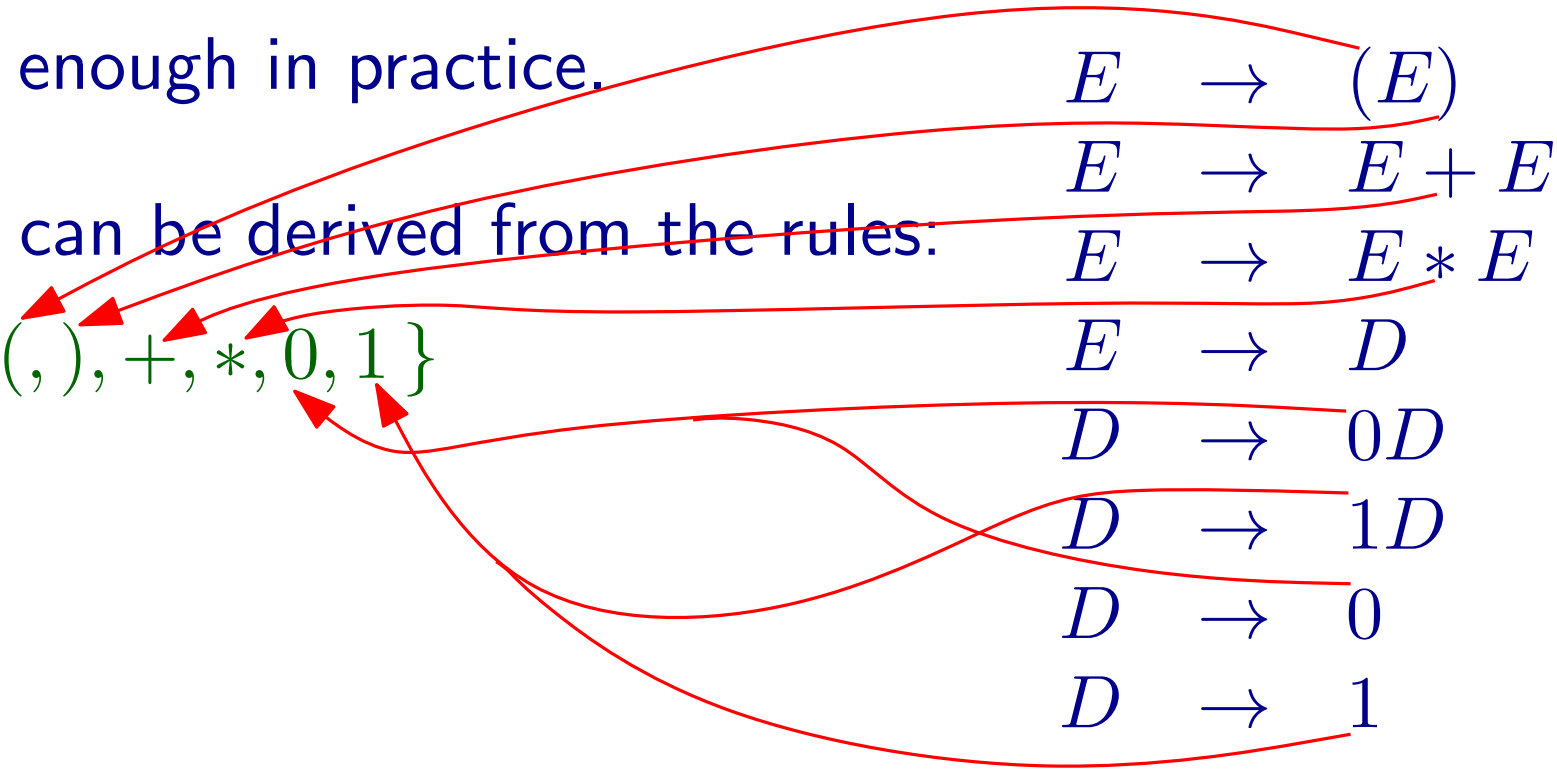
$$
\begin{aligned}
E &\rightarrow (E) \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow D \\
D &\rightarrow 0D \\
D &\rightarrow 1D \\
D &\rightarrow 0 \\
D &\rightarrow 1
\end{aligned}
$$

In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$



$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

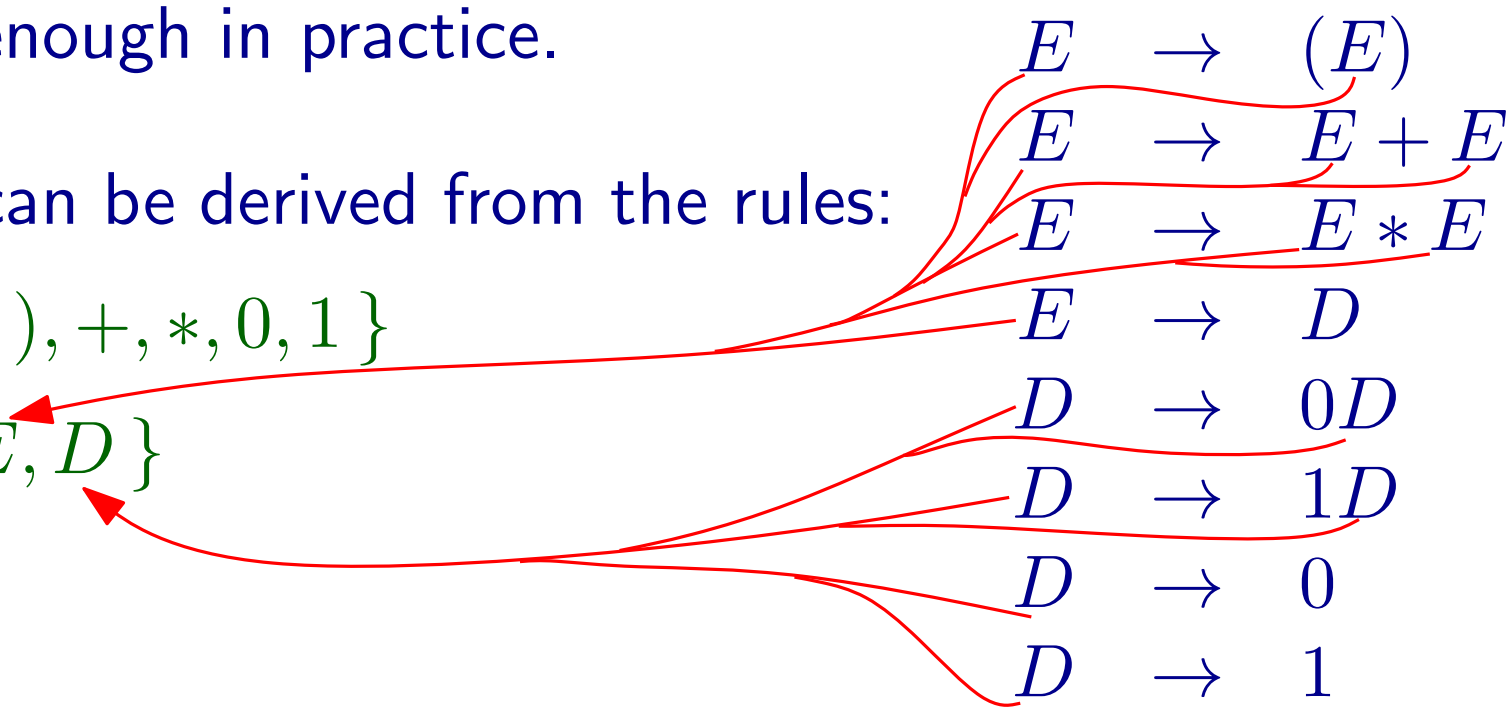
In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$

$$N = \{ E, D \}$$



In practice...

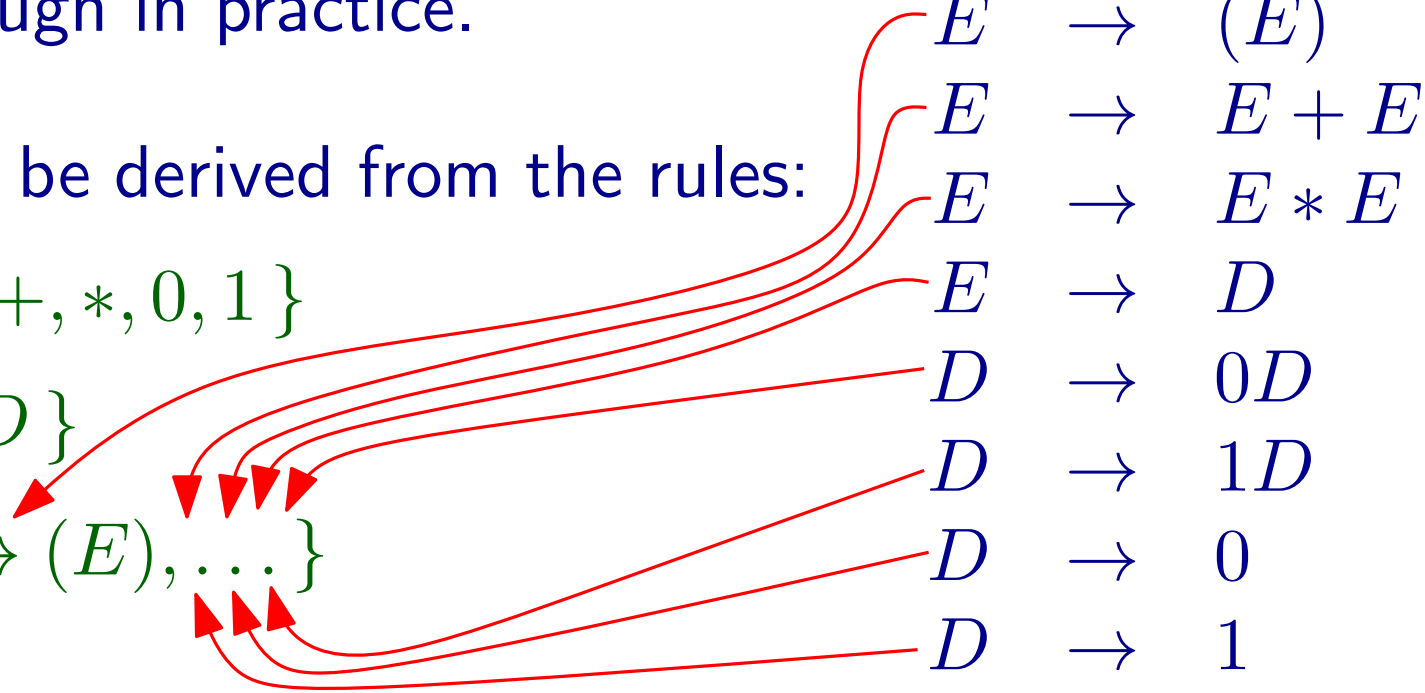
Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$

$$N = \{ E, D \}$$

$$\Pi = \{ E \to (E), \ldots \}$$



In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$

$$N = \{ E, D \}$$

$$\Pi = \{ E \to (E), \ldots \}$$

$$\begin{array}{lcl} E & \to & (E) \\ E & \to & E + E \\ E & \to & E * E \\ E & \to & D \\ D & \to & 0D \\ D & \to & 1D \\ D & \to & 0 \\ D & \to & 1 \end{array}$$

Start non-terminal: E (the LHS of the first rule)

In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$

$$N = \{ E, D \}$$

$$\Pi = \{ E \to (E), \ldots \}$$

$$\begin{array}{lcl} E & \to & (E) \\ E & \to & E + E \\ E & \to & E * E \\ E & \to & D \\ D & \to & 0D \\ D & \to & 1D \\ D & \to & 0 \\ D & \to & 1 \end{array}$$

Start non-terminal: E (the LHS of the first rule)

$$G = \langle \Sigma, N, \Pi, E \rangle$$

Derivation Tree

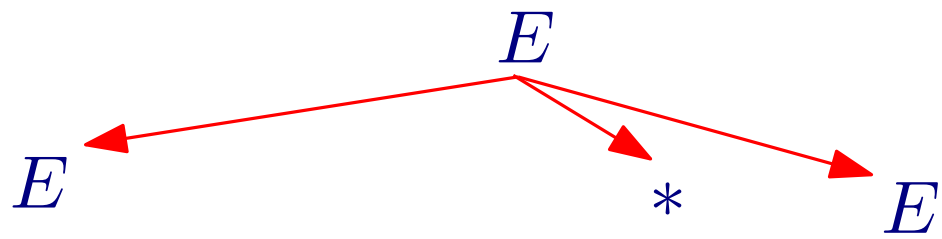
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

E

$(\quad 1 \quad + \quad 1 \quad 0 \quad) \quad * \quad 1$

Derivation Tree

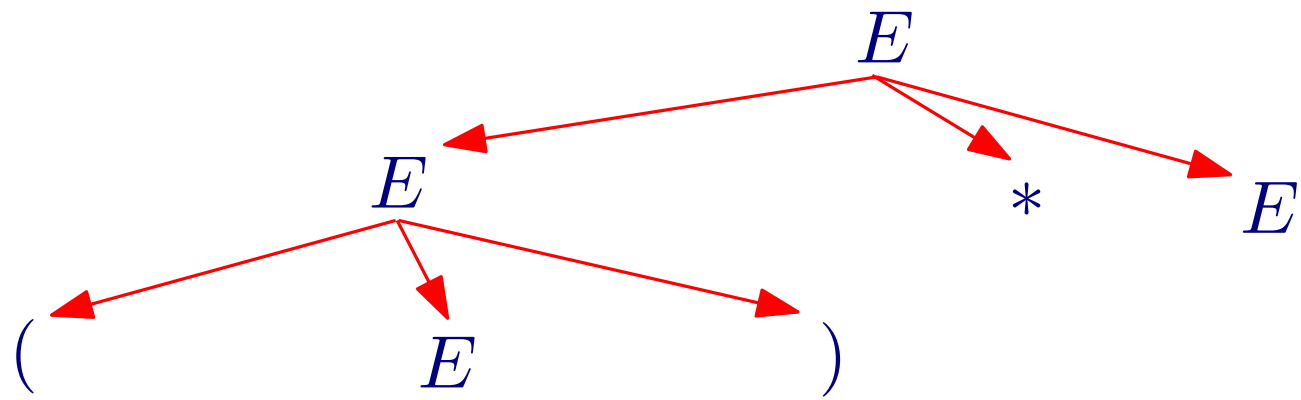
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



$$(\quad 1 \quad + \quad 1 \quad 0 \quad) \quad * \quad 1$$

Derivation Tree

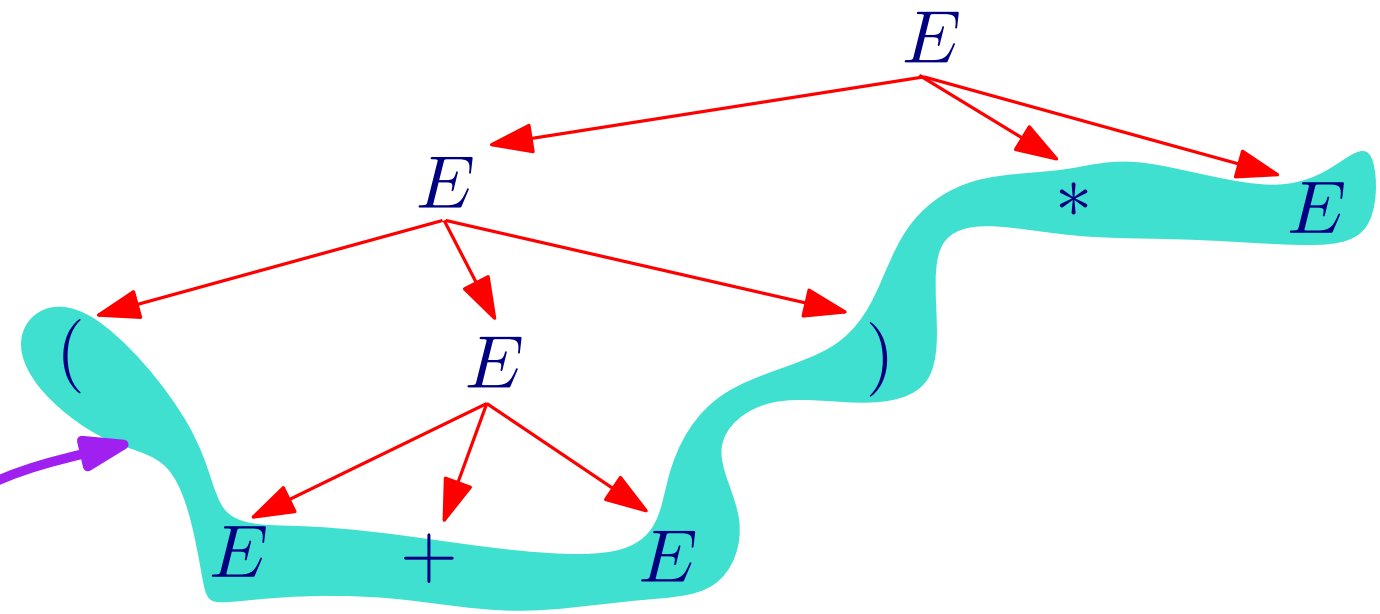
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

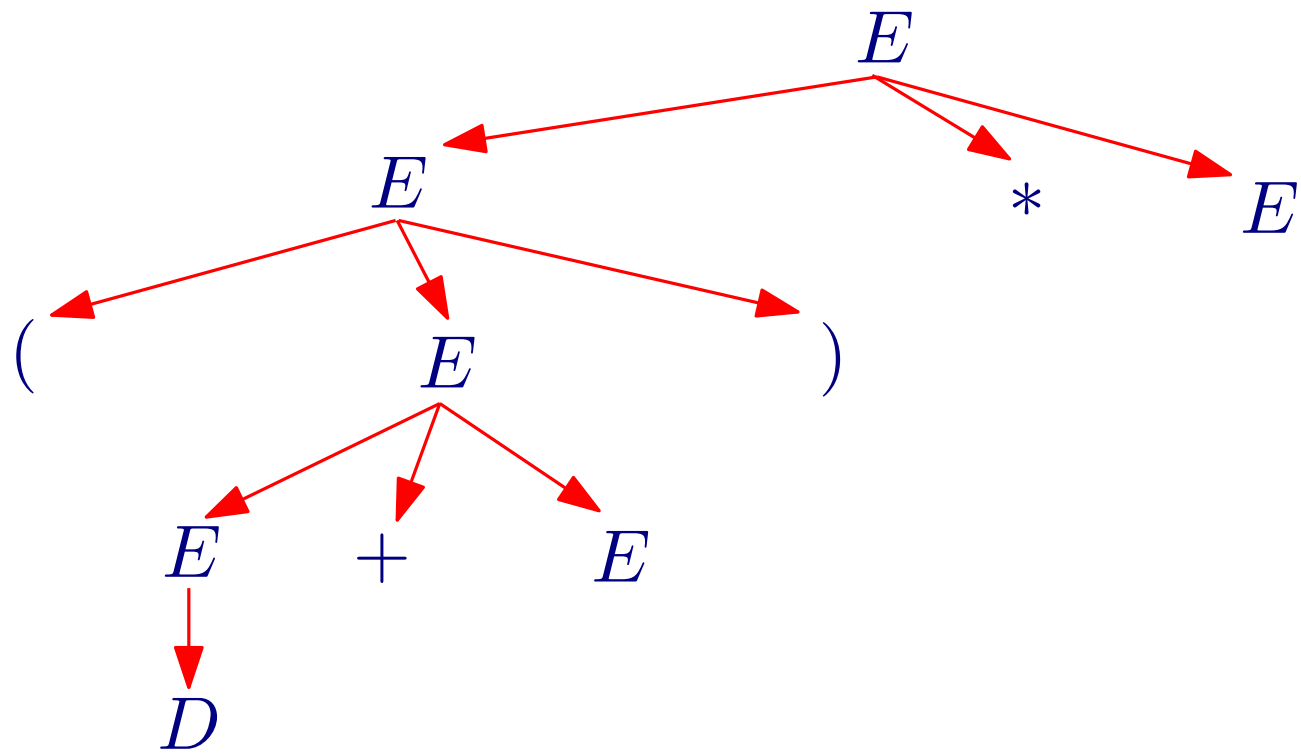


Current frontier of derivation tree: *sentential form*

matched: (1 + 1 0) * 1

Derivation Tree

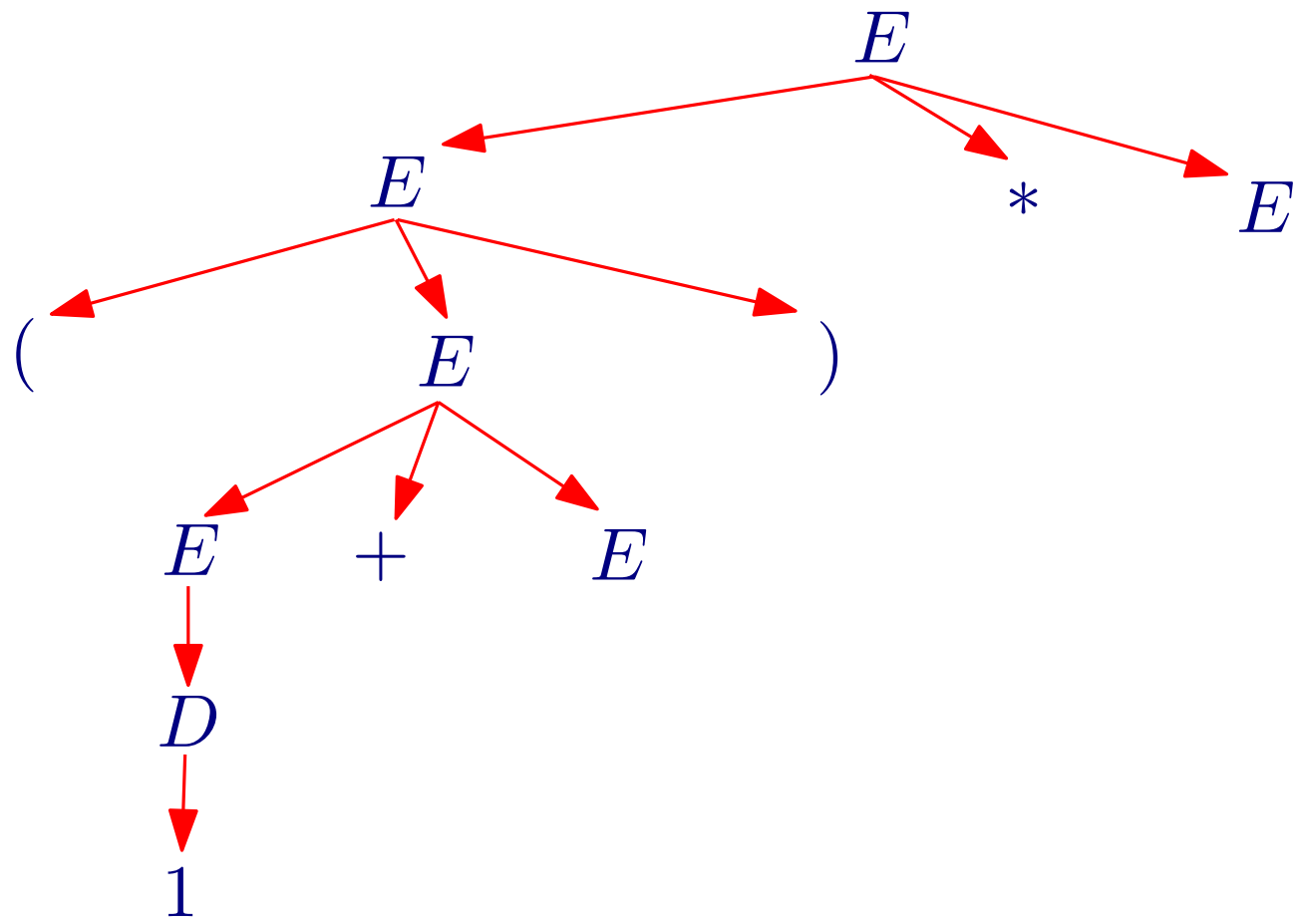
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

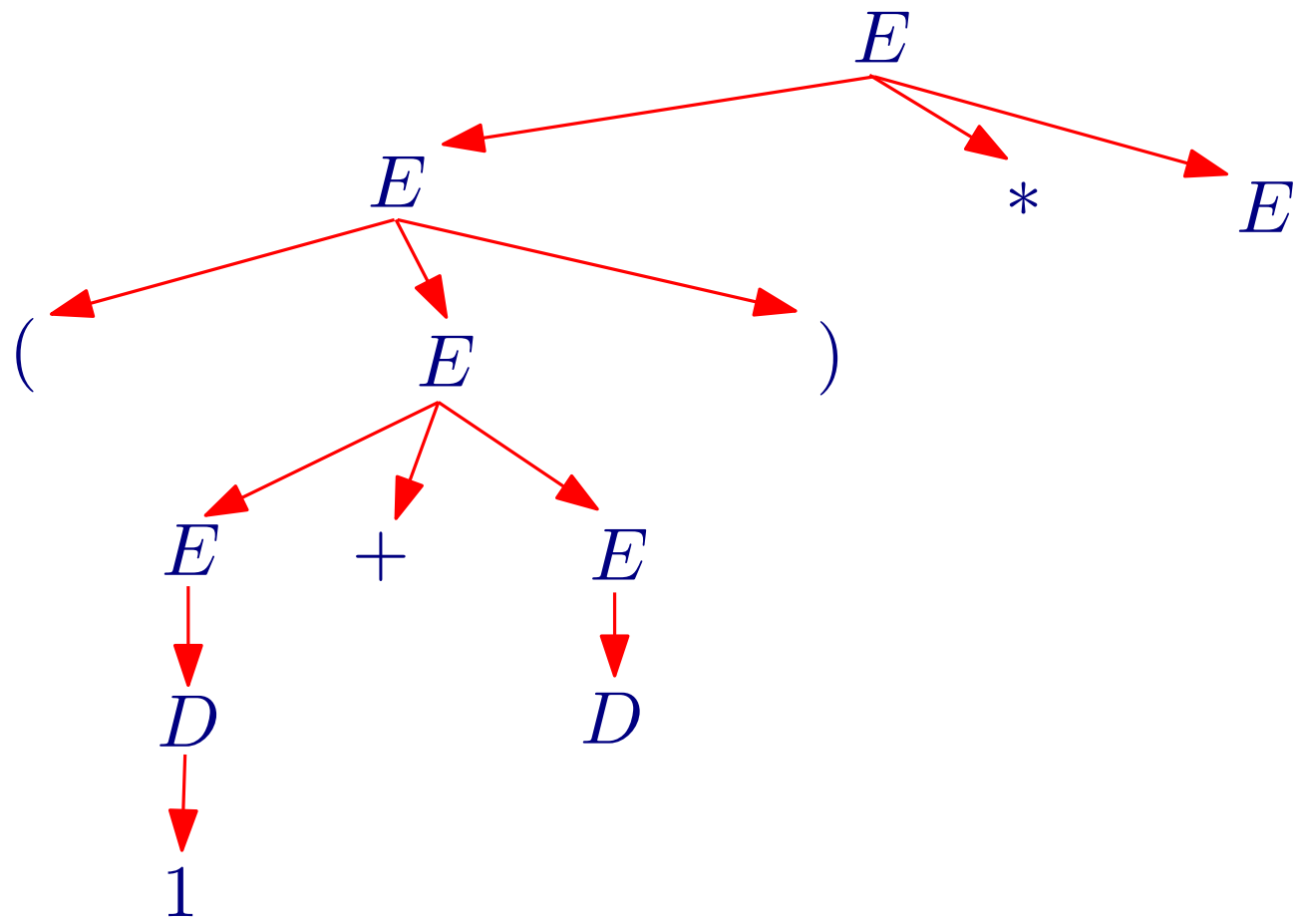
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

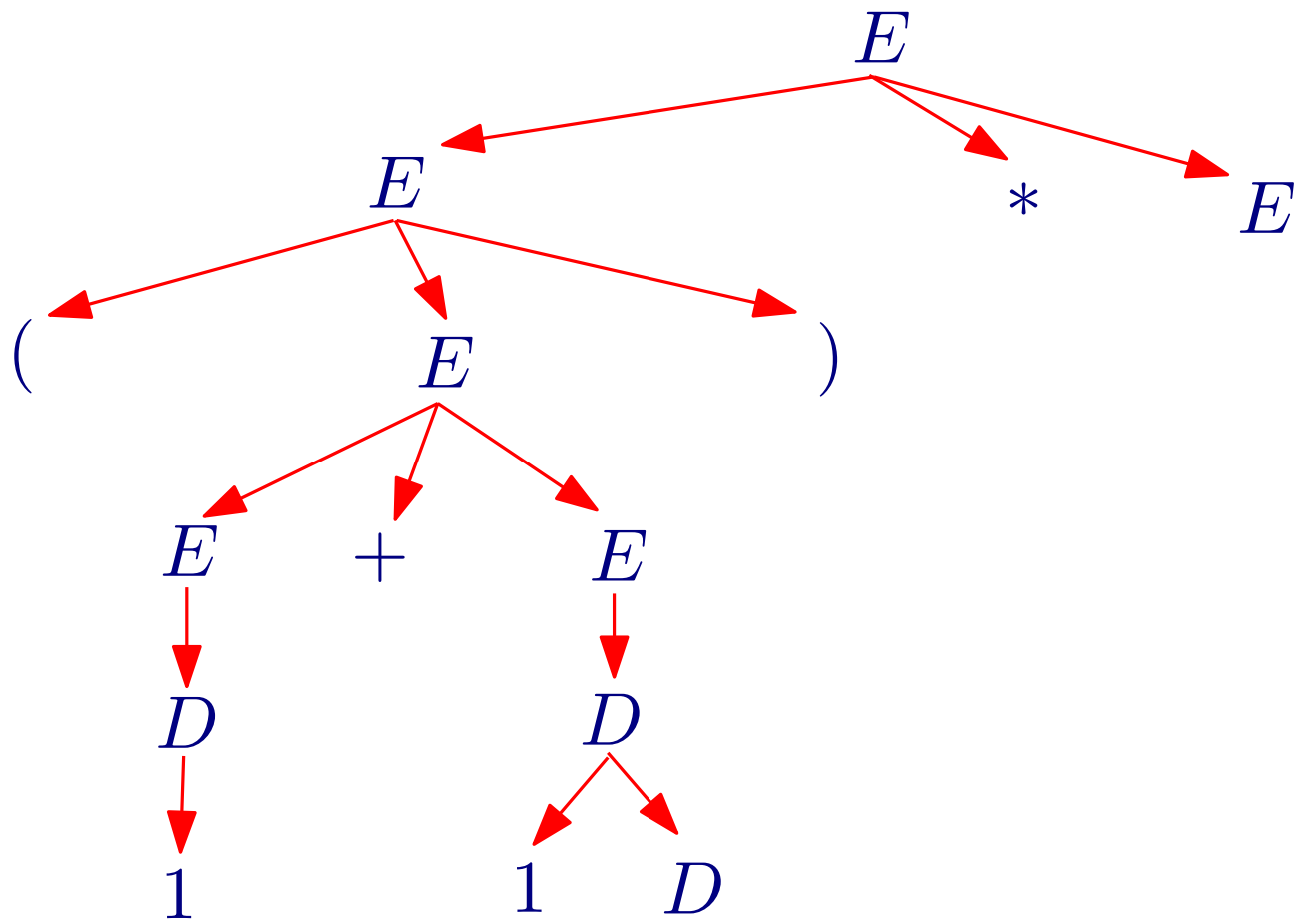
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

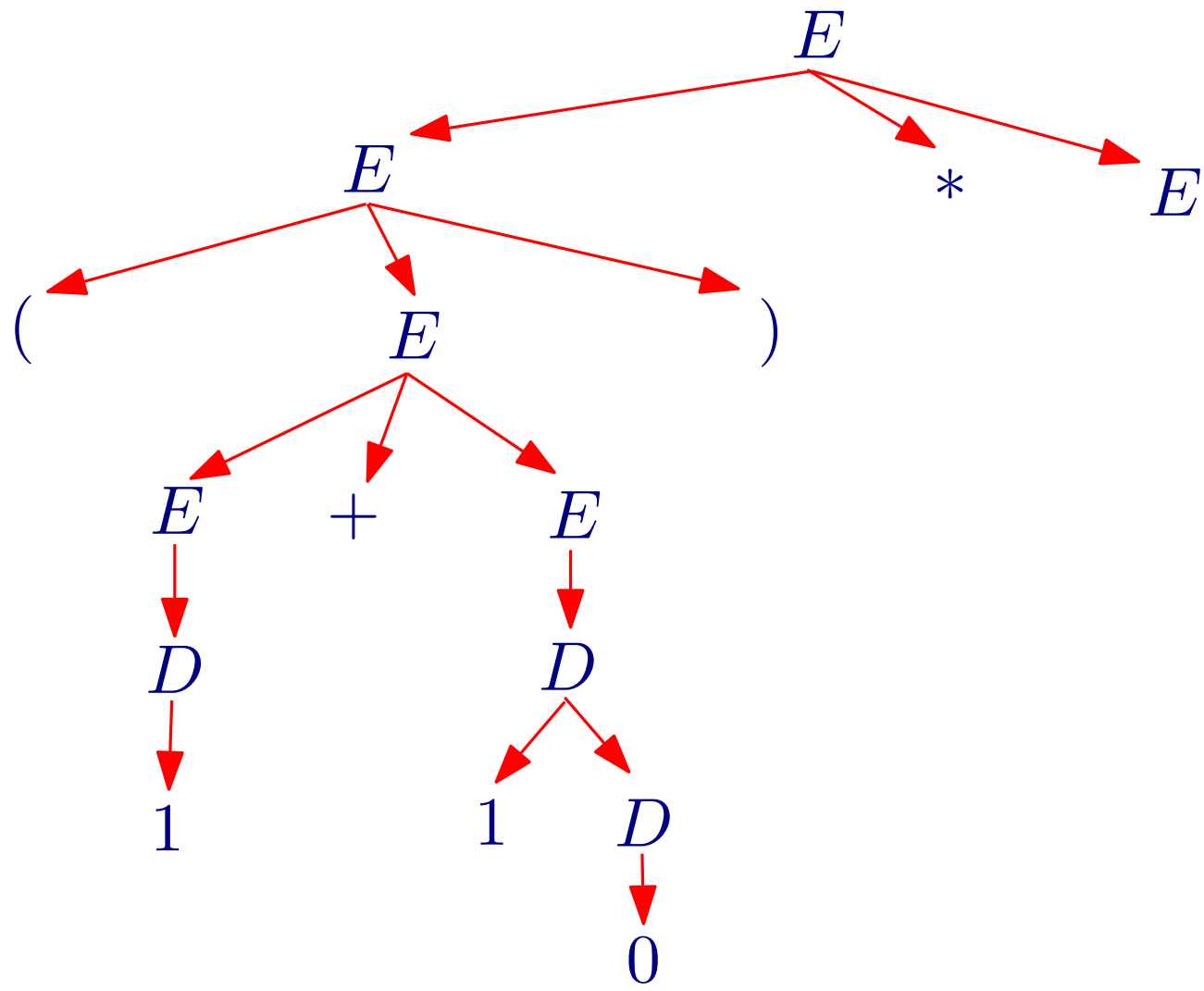
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

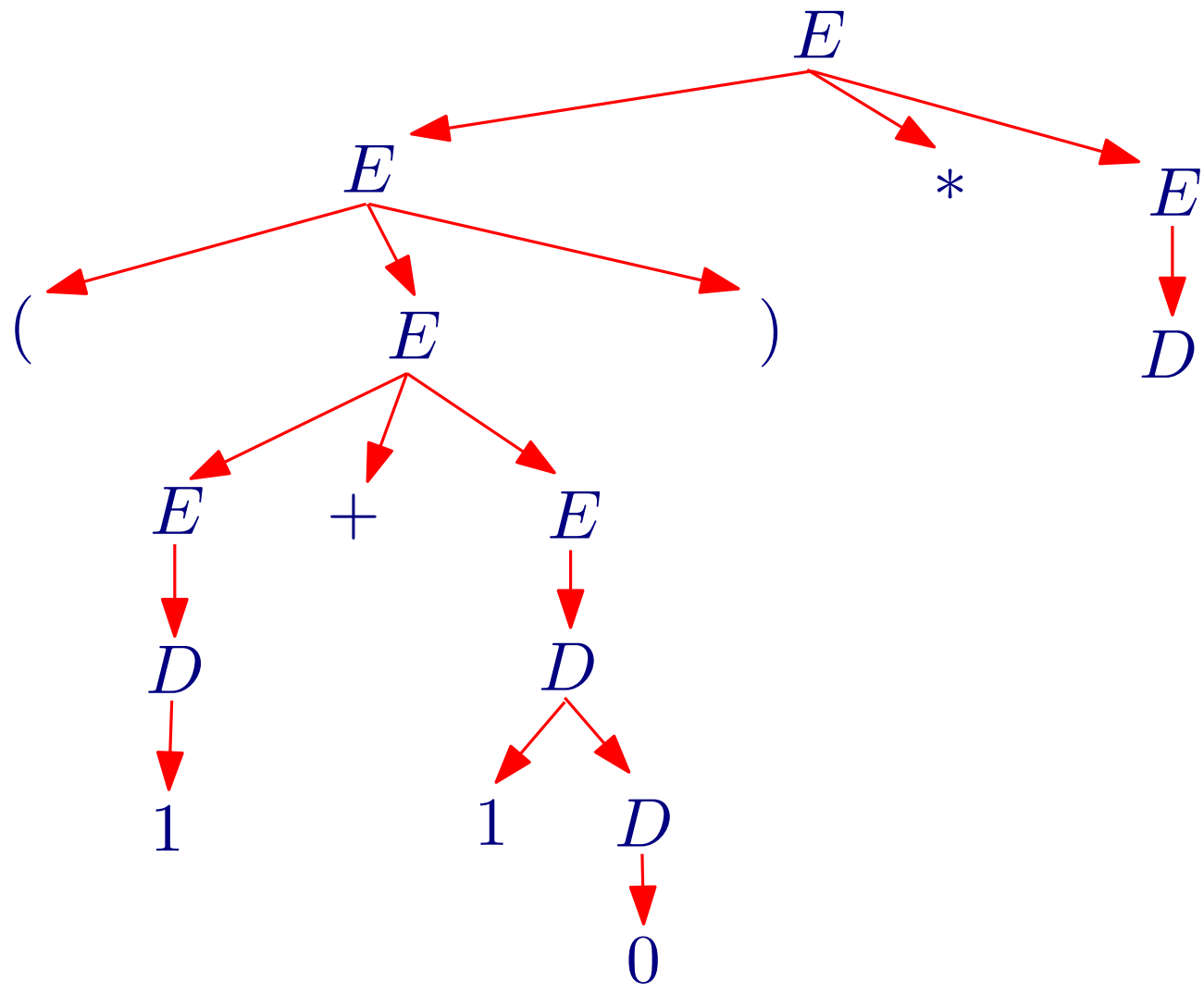
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

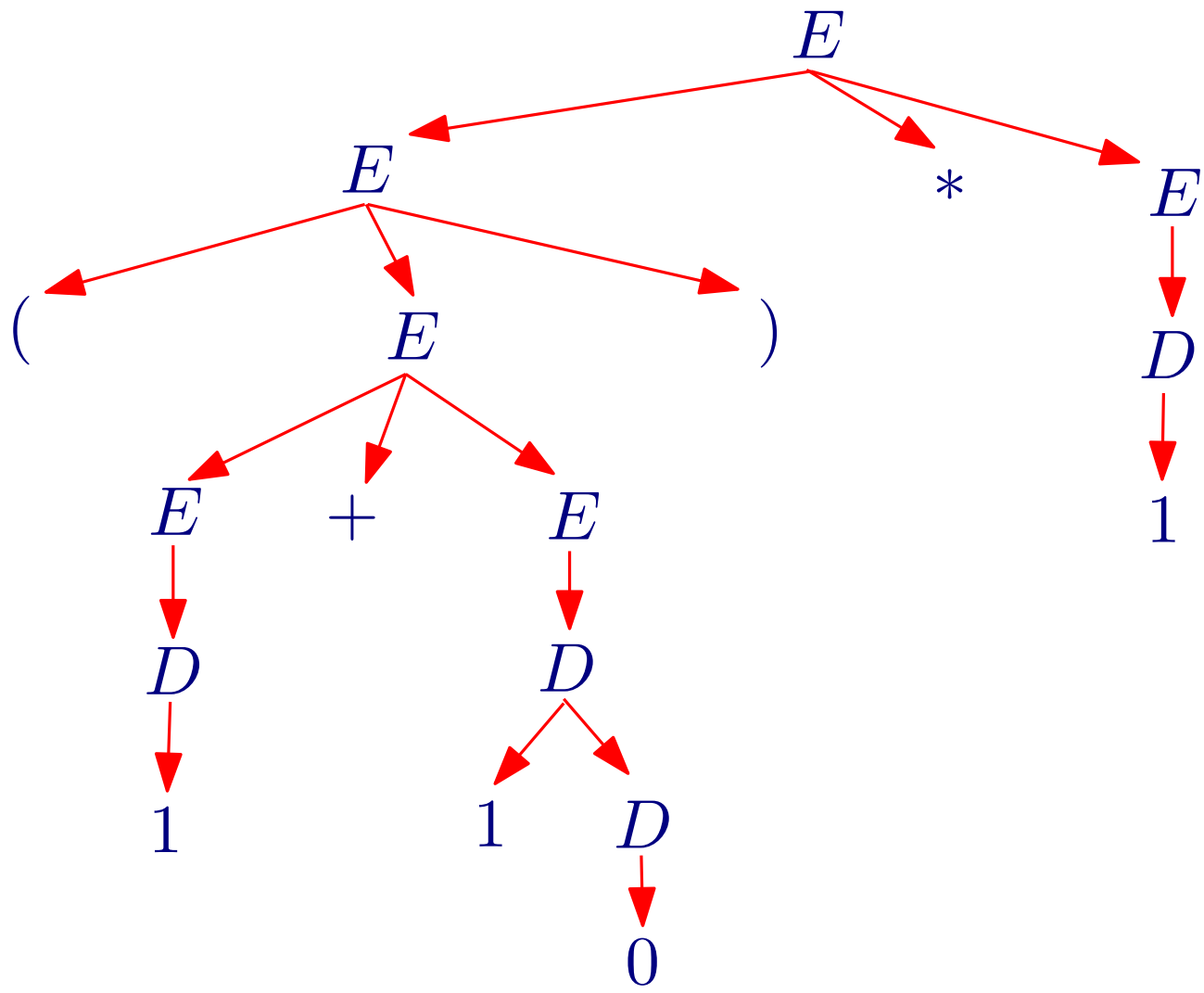
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

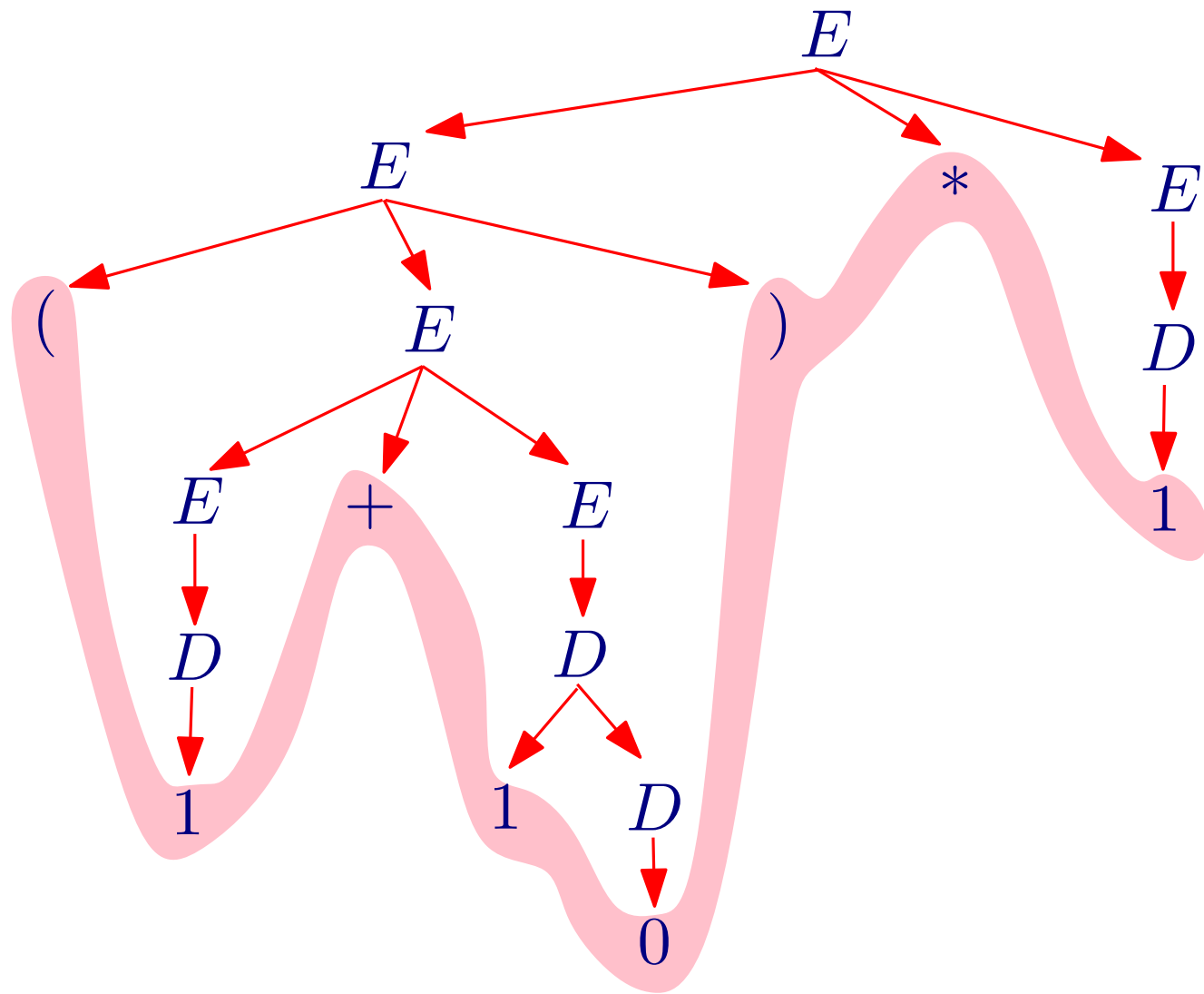
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



matched: (1 + 1 0) * 1

Derivation Tree

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$

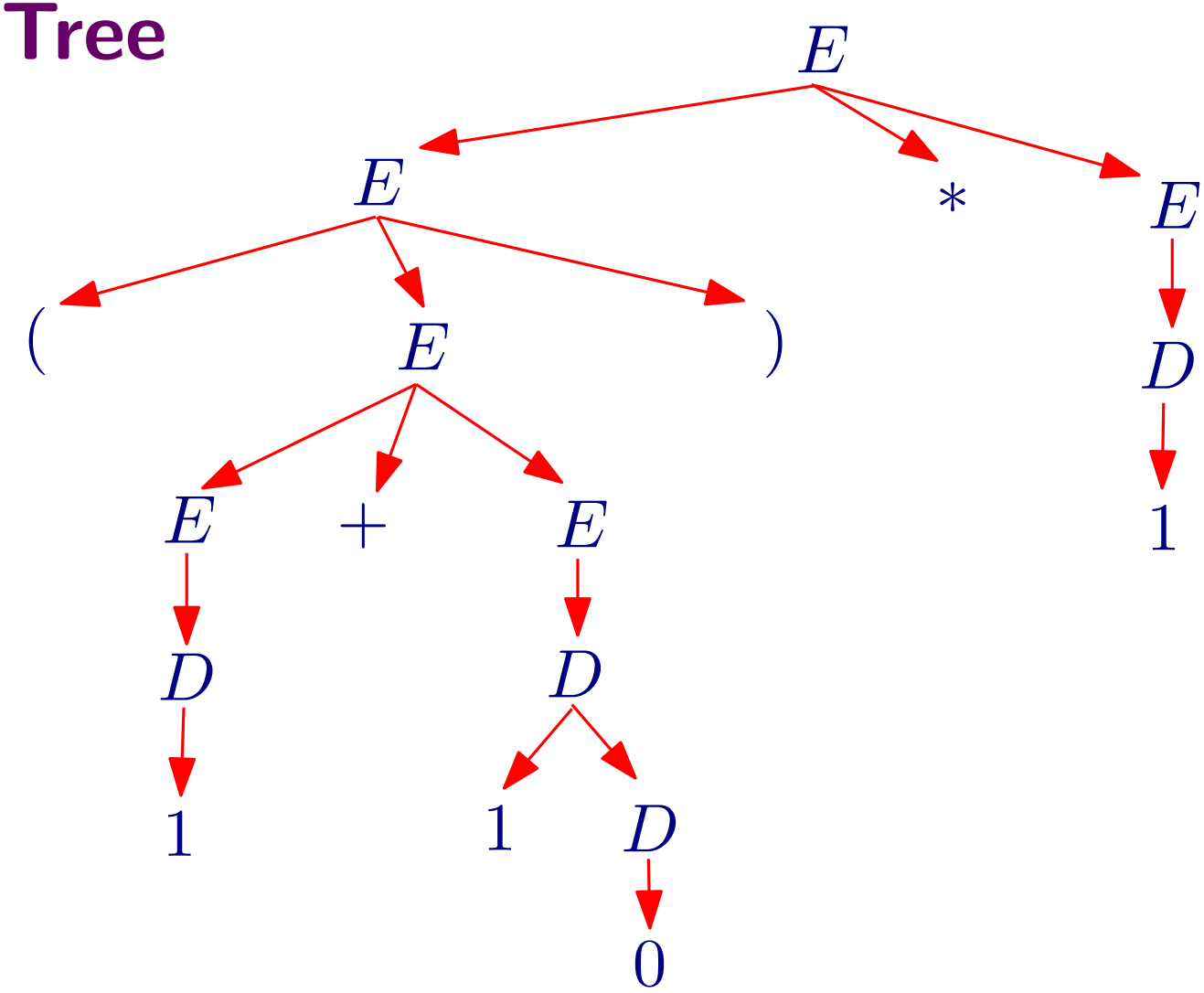


matched: (1 + 1 0) * 1

Abstract Syntax Tree

Simplified version of the parse tree:

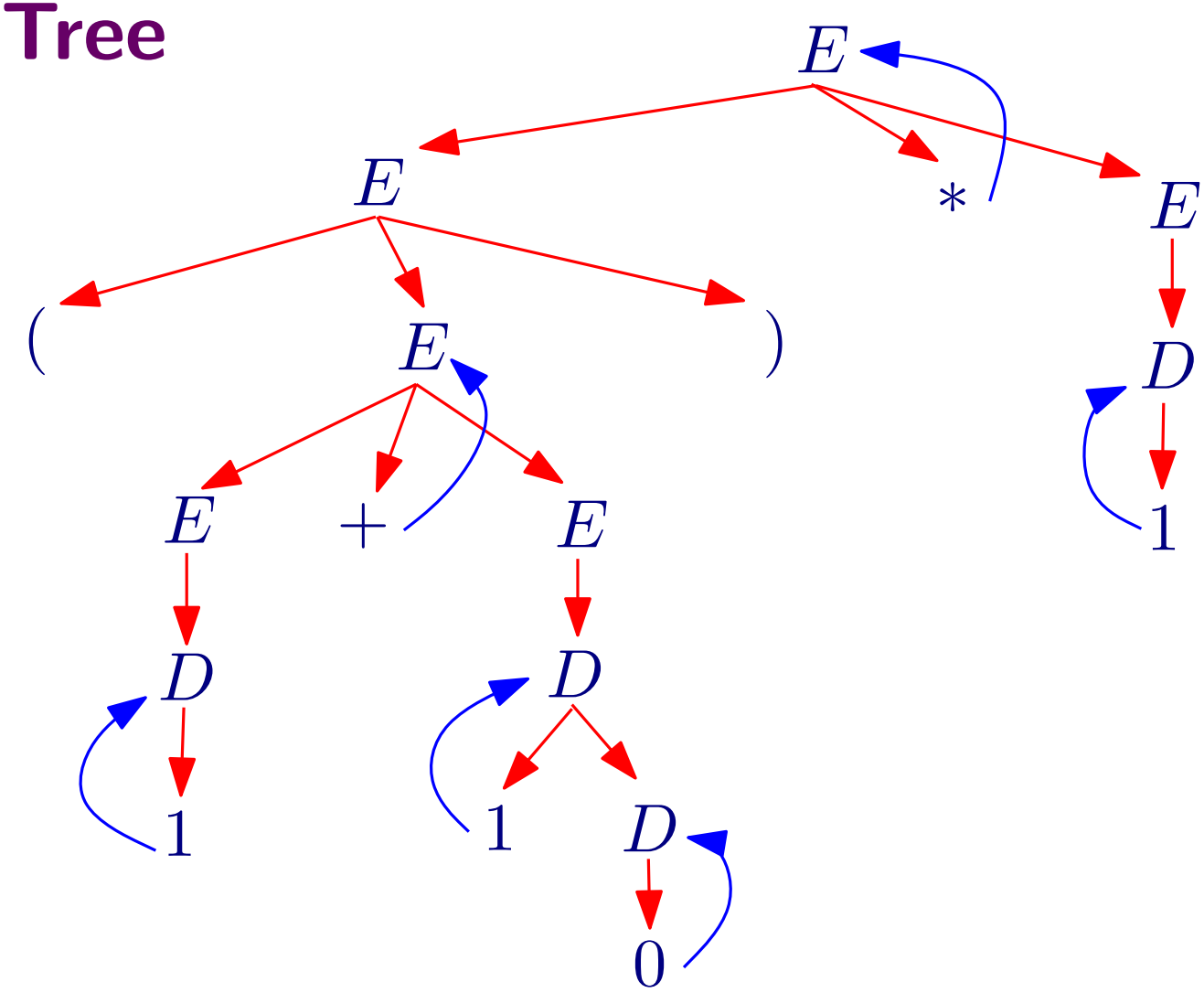
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Abstract Syntax Tree

Simplified version of the parse tree:

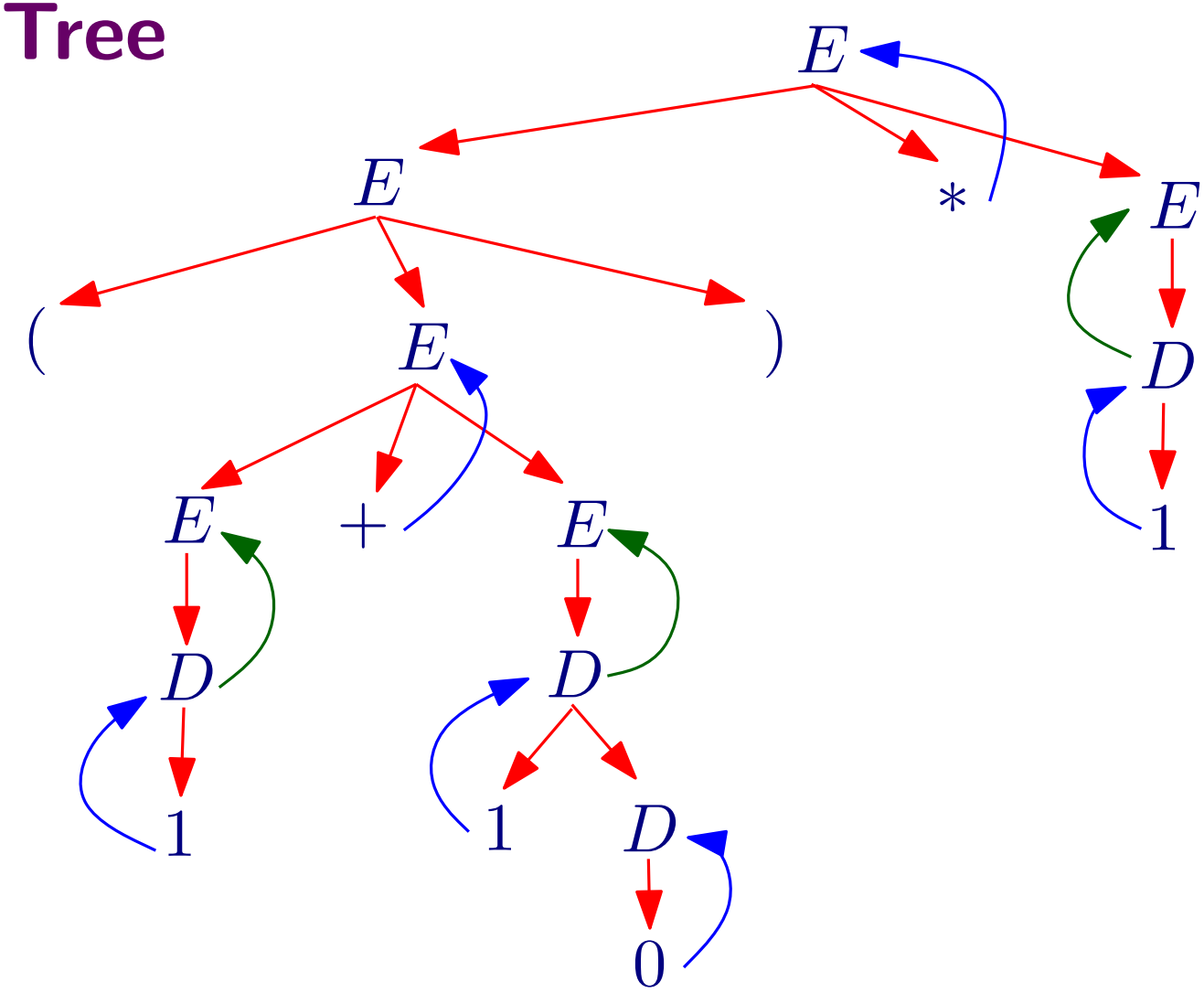
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Abstract Syntax Tree

Simplified version of the parse tree:

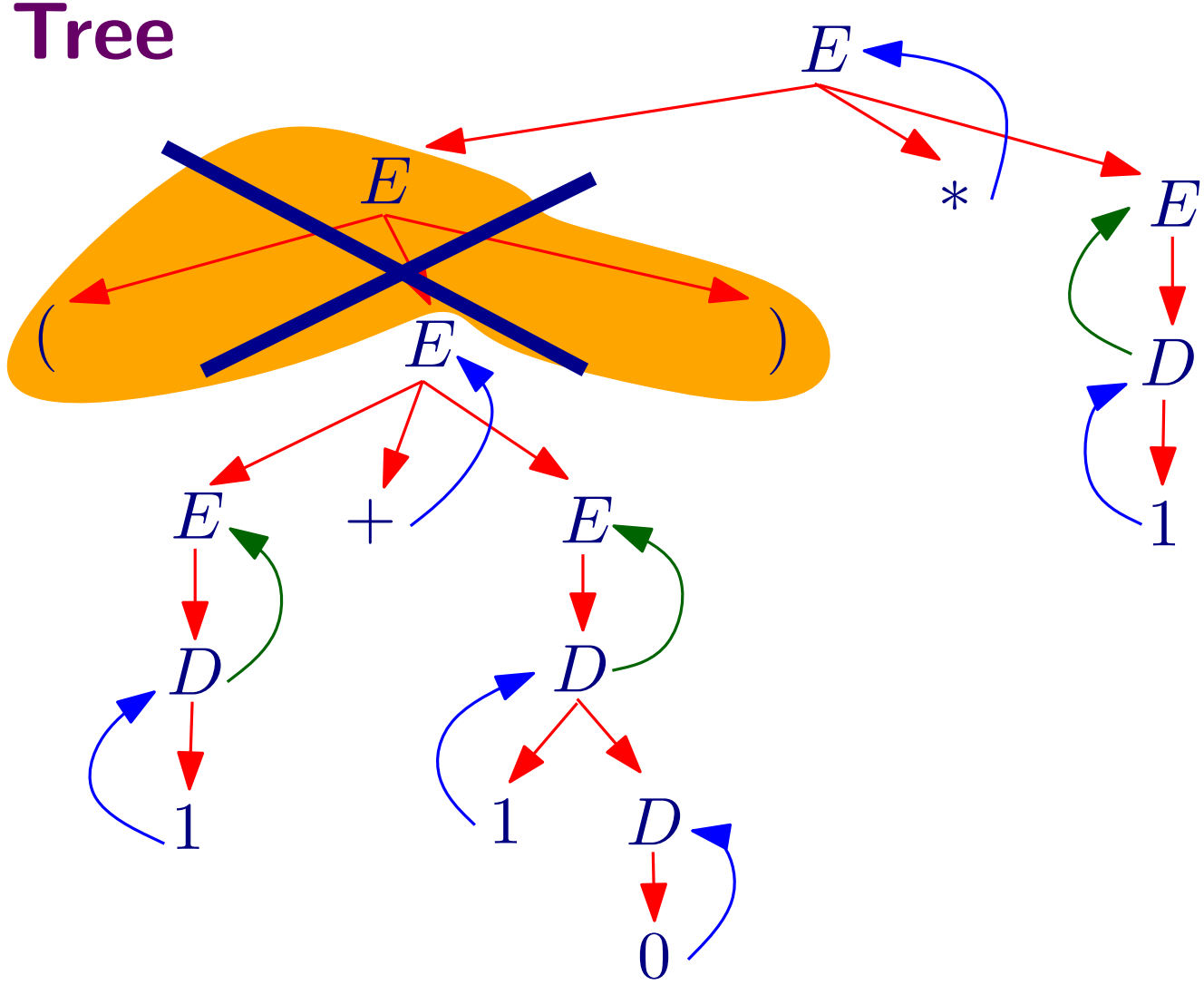
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Abstract Syntax Tree

Simplified version of the parse tree:

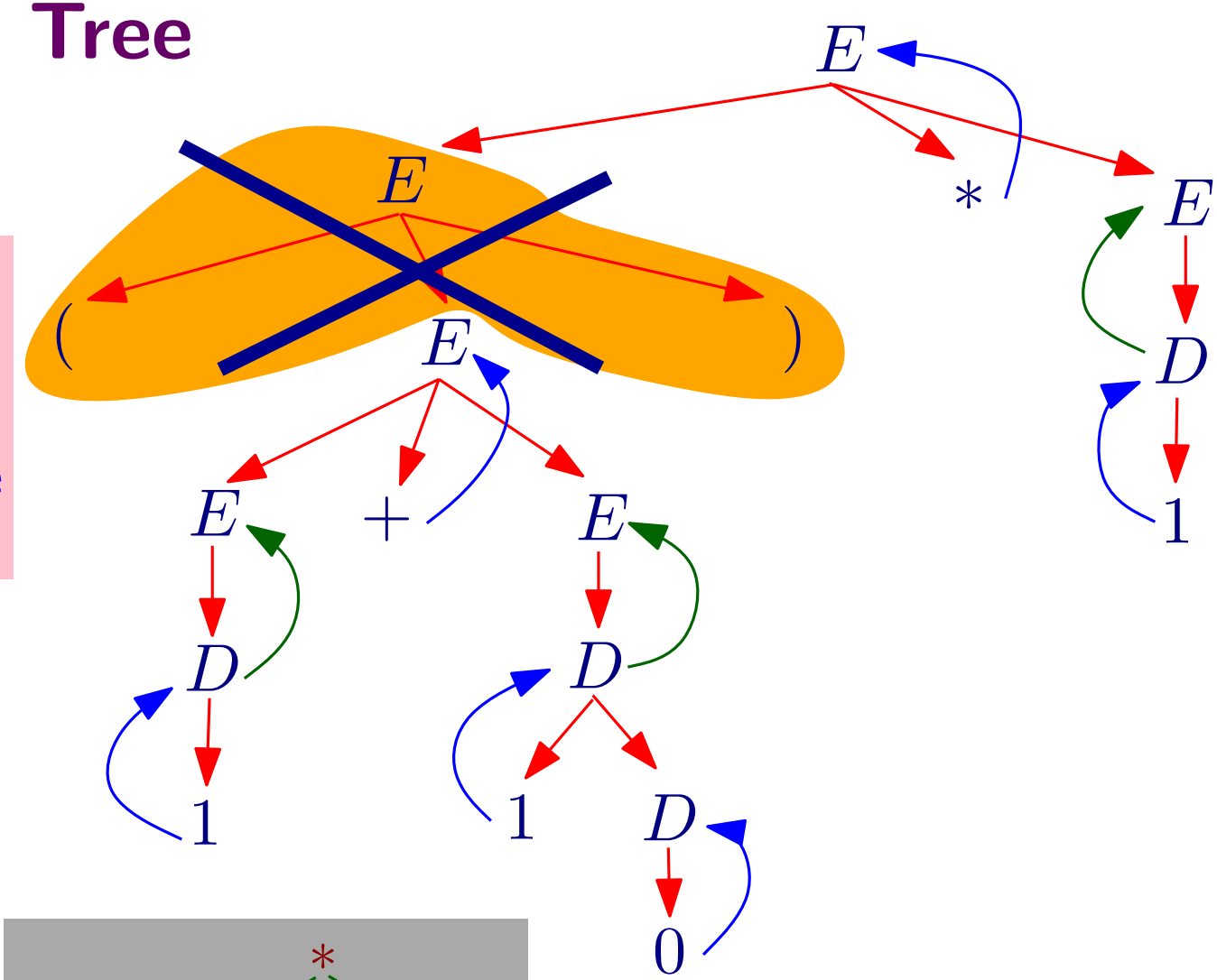
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Abstract Syntax Tree

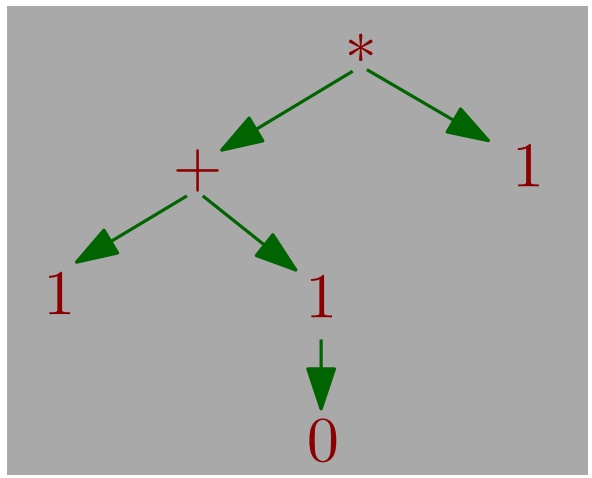
Simplified version of the parse tree:

- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Resulting tree:

Simple, but still captures the structure of the expression.



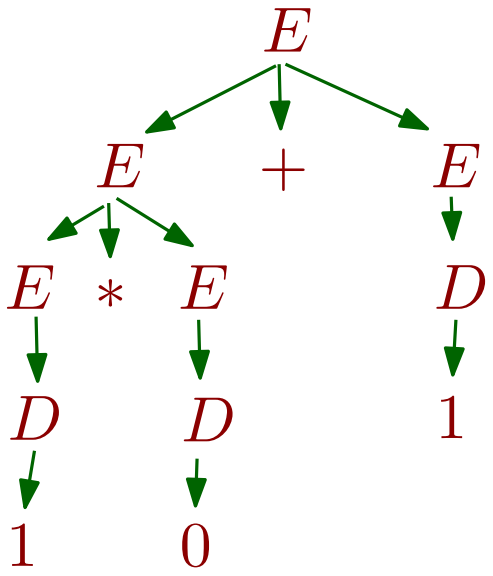
What Are Grammars Good For?

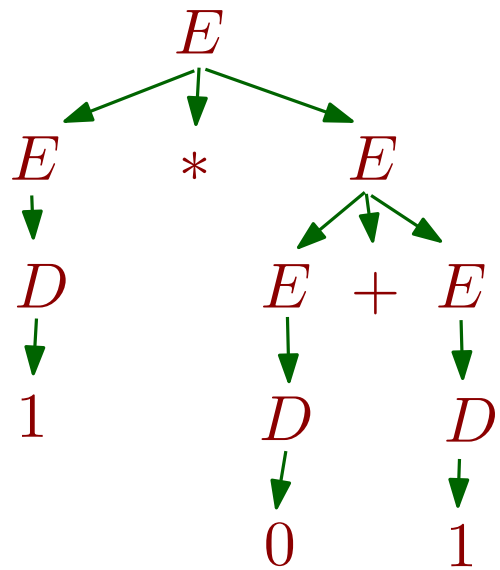
- Allow specification of (programming) languages – *generation*.
- Allow deciding whether a string belongs to the language or not – *acceptance*.
- Allow *syntactic analysis*.
- **Syntactic analysis:** building the Abstract Syntax Tree from a string or program.
- The AST can be used by compilers, program analyzers, and other code manipulators.

Ambiguity

Non-unique parse trees: *ambiguous grammar*

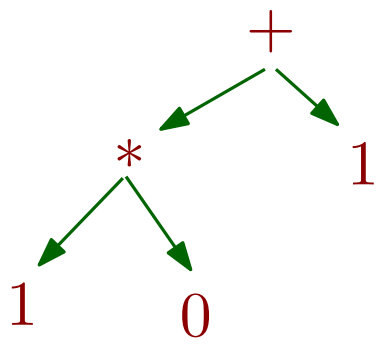
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



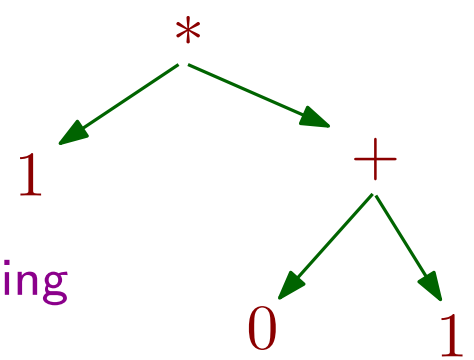


1 * 0 + 1

1 * 0 + 1



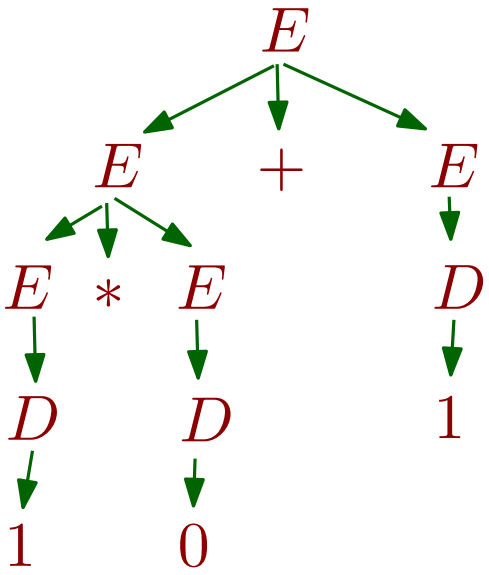
Corresponding ASTs

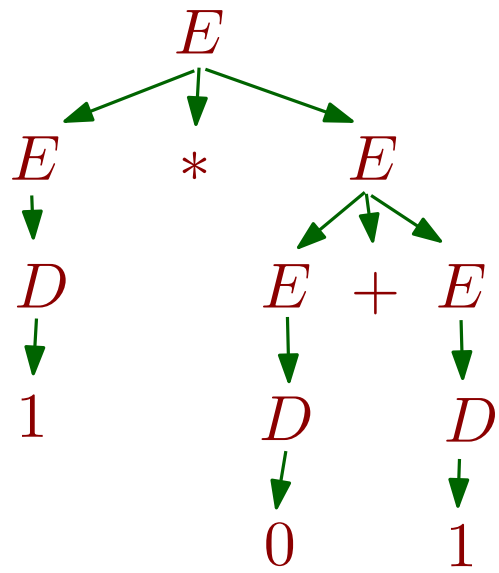


Ambiguity

Non-unique parse trees: *ambiguous grammar*

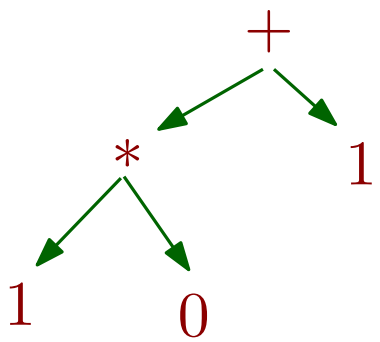
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



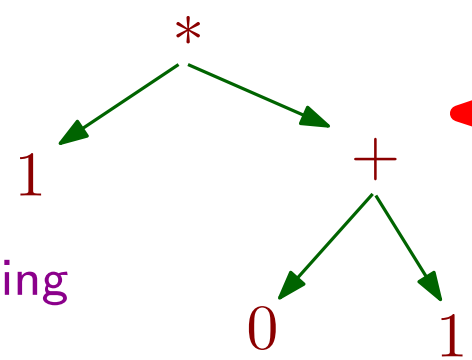


1 * 0 + 1

1 * 0 + 1



Corresponding ASTs



Ambiguous precedence of operators!

Ambiguity Should Be Avoided!

- An ambiguous grammar can always be replaced by a non-ambiguous one.
- Ambiguous grammars have fewer rules, but tend to capture less of the language's structure.
- Precedence and associativity of operators is crucial to structure of languages, and should be captured in the grammar.
- Languages with non-ambiguous grammars can be parsed more efficiently.

Backus-Naur Form

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

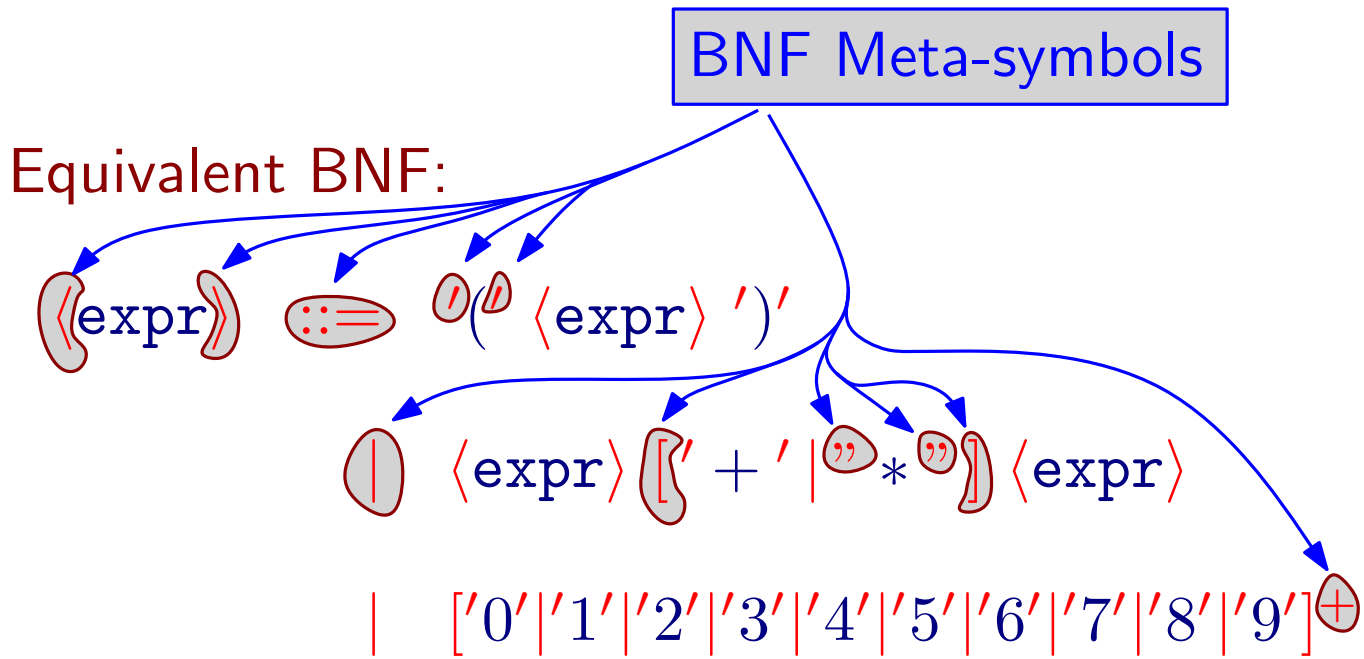
Equivalent BNF:

$$
\begin{array}{lcl}
\langle \texttt{expr} \rangle & ::= & \text{'('} \ \langle \texttt{expr} \rangle \ \text{')'} \\
& | & \langle \texttt{expr} \rangle \ [\text{'+'} \mid \text{"*"}] \ \langle \texttt{expr} \rangle \\
& | & [\text{'0'} \mid \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'}]^{+}
\end{array}
$$

Backus-Naur Form

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$



Backus-Naur Form

Non-terminals are enclosed in angle brackets

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Equivalent BNF:

$$
\begin{array}{lcl}
\langle \texttt{expr} \rangle & ::= & \text{'('} \ \langle \texttt{expr} \rangle \ \text{')'} \\
& | & \langle \texttt{expr} \rangle \ [\text{'+'} \,|\, \text{"*"}] \ \langle \texttt{expr} \rangle \\
& | & [\text{'0'} \,|\, \text{'1'} \,|\, \text{'2'} \,|\, \text{'3'} \,|\, \text{'4'} \,|\, \text{'5'} \,|\, \text{'6'} \,|\, \text{'7'} \,|\, \text{'8'} \,|\, \text{'9'}]^{+}
\end{array}
$$

Backus-Naur Form

Terminals are enclosed in simple or double quotes.

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Equivalent BNF:

$$
\begin{array}{lcl}
\langle \texttt{expr} \rangle & ::= & '(' \ \langle \texttt{expr} \rangle \ ')' \\
 & | & \langle \texttt{expr} \rangle \ [' + ' | \text{" } * \text{ "}] \ \langle \texttt{expr} \rangle \\
 & | & ['0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9']^{+}
\end{array}
$$

Single quote terminal: "'"
Double quote terminal: '"'

Backus-Naur Form

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Equivalent BNF:

$$
\begin{array}{lcl}
\langle \texttt{expr} \rangle & ::= & \text{'('} \ \langle \texttt{expr} \rangle \ \text{')'} \\
& | & \langle \texttt{expr} \rangle \ [\text{'+'} \,|\, \text{"*"}] \ \langle \texttt{expr} \rangle \\
& | & [\text{'0'}|\text{'1'}|\text{'2'}|\text{'3'}|\text{'4'}|\text{'5'}|\text{'6'}|\text{'7'}|\text{'8'}|\text{'9'}]^{+}
\end{array}
$$

Backus-Naur Form

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Grouping

Equivalent BNF:

$$
\begin{array}{lcl}
\langle \texttt{expr} \rangle & ::= & \texttt{'('} \ \langle \texttt{expr} \rangle \ \texttt{')'} \\
& | & \langle \texttt{expr} \rangle \ [\texttt{'+'} \,|\, \texttt{"*"}] \ \langle \texttt{expr} \rangle \\
& | & [\texttt{'0'} \,|\, \texttt{'1'} \,|\, \texttt{'2'} \,|\, \texttt{'3'} \,|\, \texttt{'4'} \,|\, \texttt{'5'} \,|\, \texttt{'6'} \,|\, \texttt{'7'} \,|\, \texttt{'8'} \,|\, \texttt{'9'}]^{+}
\end{array}
$$

Backus-Naur Form

Original grammar:

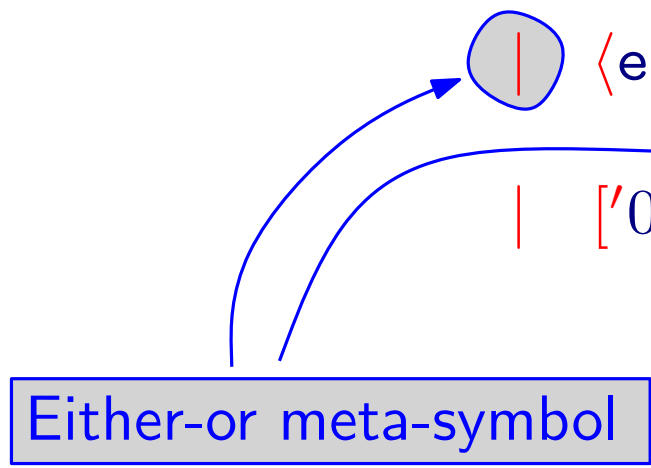
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Equivalent BNF:

$\langle \texttt{expr} \rangle$ ::= '(' $\langle \texttt{expr} \rangle$ ')'

| $\langle \texttt{expr} \rangle$ [' + '|" * "] $\langle \texttt{expr} \rangle$

| ['0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9']$^+$



Backus-Naur Form

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Factorization

Equivalent BNF:

$$
\begin{array}{lcl}
\langle \texttt{expr} \rangle & ::= & \texttt{'('} \ \langle \texttt{expr} \rangle \ \texttt{')'} \\
& | & \langle \texttt{expr} \rangle \ [\texttt{'+'} \,|\, \texttt{"*"}] \ \langle \texttt{expr} \rangle \\
& | & [\texttt{'0'}|\texttt{'1'}|\texttt{'2'}|\texttt{'3'}|\texttt{'4'}|\texttt{'5'}|\texttt{'6'}|\texttt{'7'}|\texttt{'8'}|\texttt{'9'}]^{+}
\end{array}
$$

Backus-Naur Form

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Equivalent BNF:

$$
\begin{array}{lcl}
\langle \texttt{expr} \rangle & ::= & \text{'('} \ \langle \texttt{expr} \rangle \ \text{')'} \\
& | & \langle \texttt{expr} \rangle \ [\text{'+'} \mid \text{"*"}] \ \langle \texttt{expr} \rangle \\
& | & [\text{'0'} \mid \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'}]^{+}
\end{array}
$$

Iteration

One or more repetitions

Backus-Naur Form

Original grammar:

$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 2D \\
D & \rightarrow & 3D \\
D & \rightarrow & 4D \\
D & \rightarrow & 5D \\
D & \rightarrow & 6D \\
D & \rightarrow & 7D \\
D & \rightarrow & 8D \\
D & \rightarrow & 9D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1 \\
D & \rightarrow & 2 \\
D & \rightarrow & 3 \\
D & \rightarrow & 4 \\
D & \rightarrow & 5 \\
D & \rightarrow & 6 \\
D & \rightarrow & 7 \\
D & \rightarrow & 8 \\
D & \rightarrow & 9
\end{array}
$$

Equivalent BNF:

$$
\begin{array}{rcl}
\langle \texttt{expr} \rangle & ::= & \texttt{'('} \ \langle \texttt{expr} \rangle \ \texttt{')'} \\
& | & \langle \texttt{expr} \rangle \ [\texttt{'+'} \,|\, \texttt{"*"}] \ \langle \texttt{expr} \rangle \\
& | & [\texttt{'0'}|\texttt{'1'}|\texttt{'2'}|\texttt{'3'}|\texttt{'4'}|\texttt{'5'}|\texttt{'6'}|\texttt{'7'}|\texttt{'8'}|\texttt{'9'}]\oplus
\end{array}
$$

Iteration

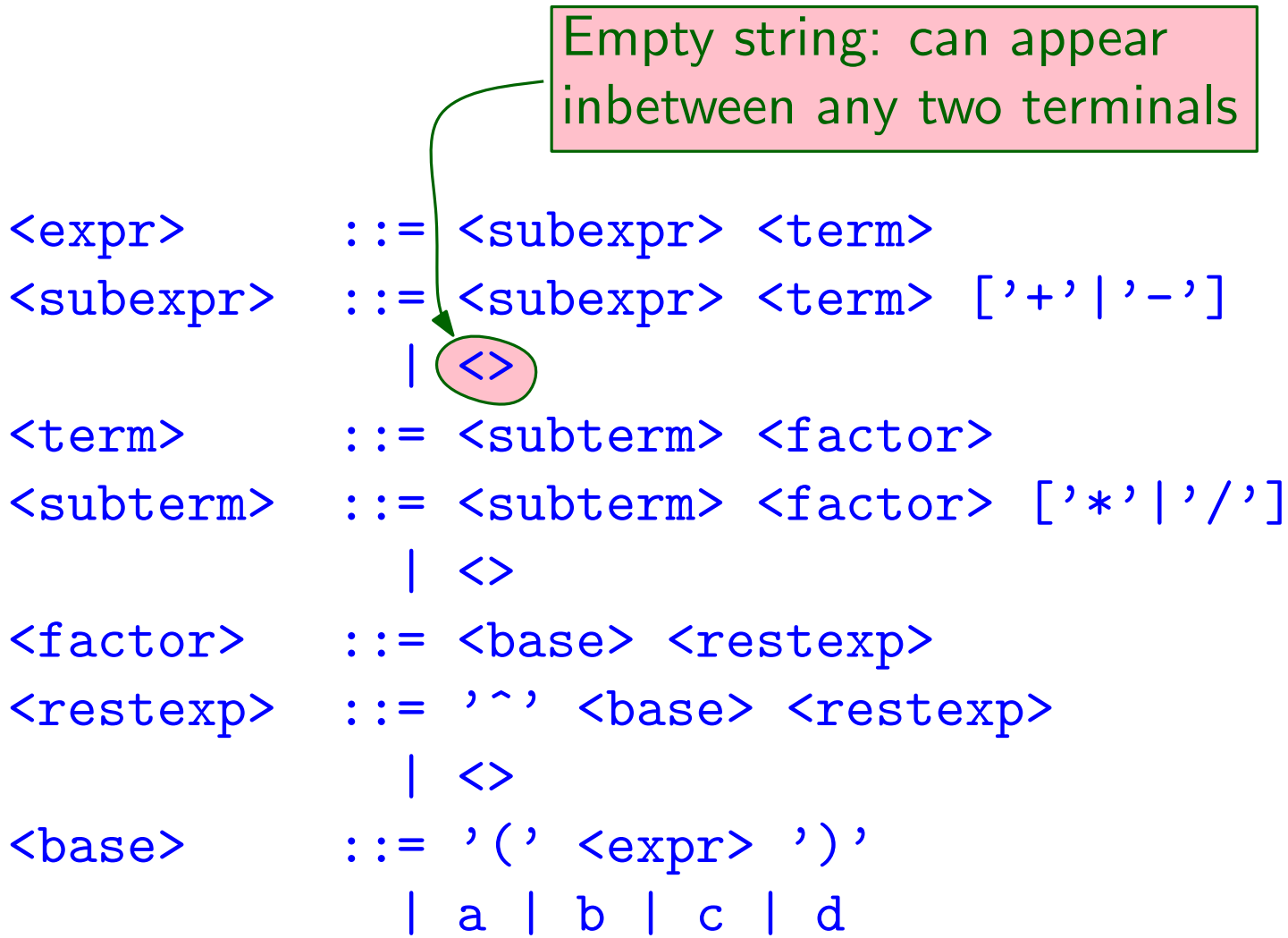
One or more repetitions

For 0 or more repetitions, use *

A Non-Ambiguous Grammar for Expressions

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

A Non-Ambiguous Grammar for Expressions

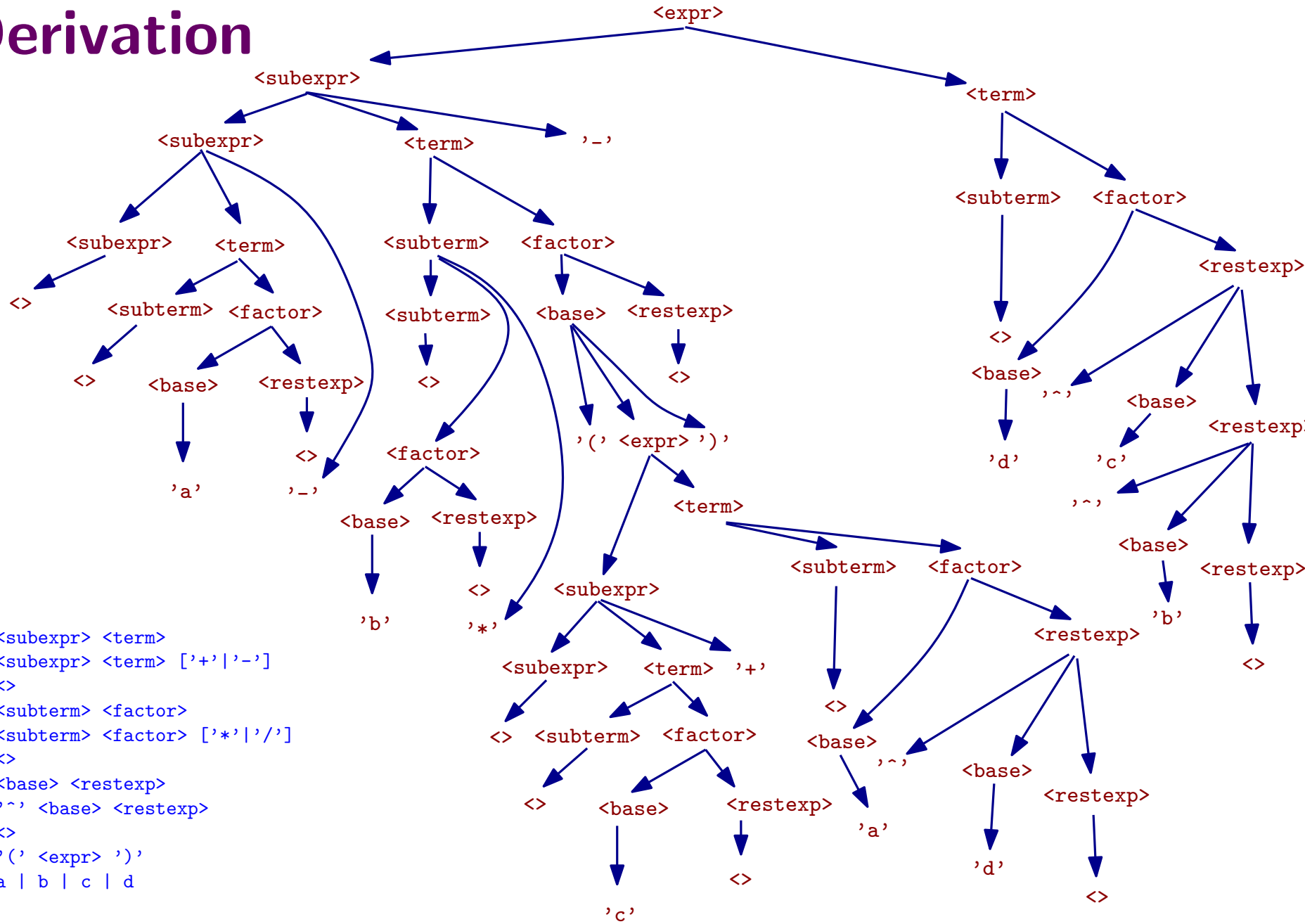


A Non-Ambiguous Grammar for Expressions

Compress the spec, but information about associativity is lost!

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= [ <term> ['+'|'-'] ]*

<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

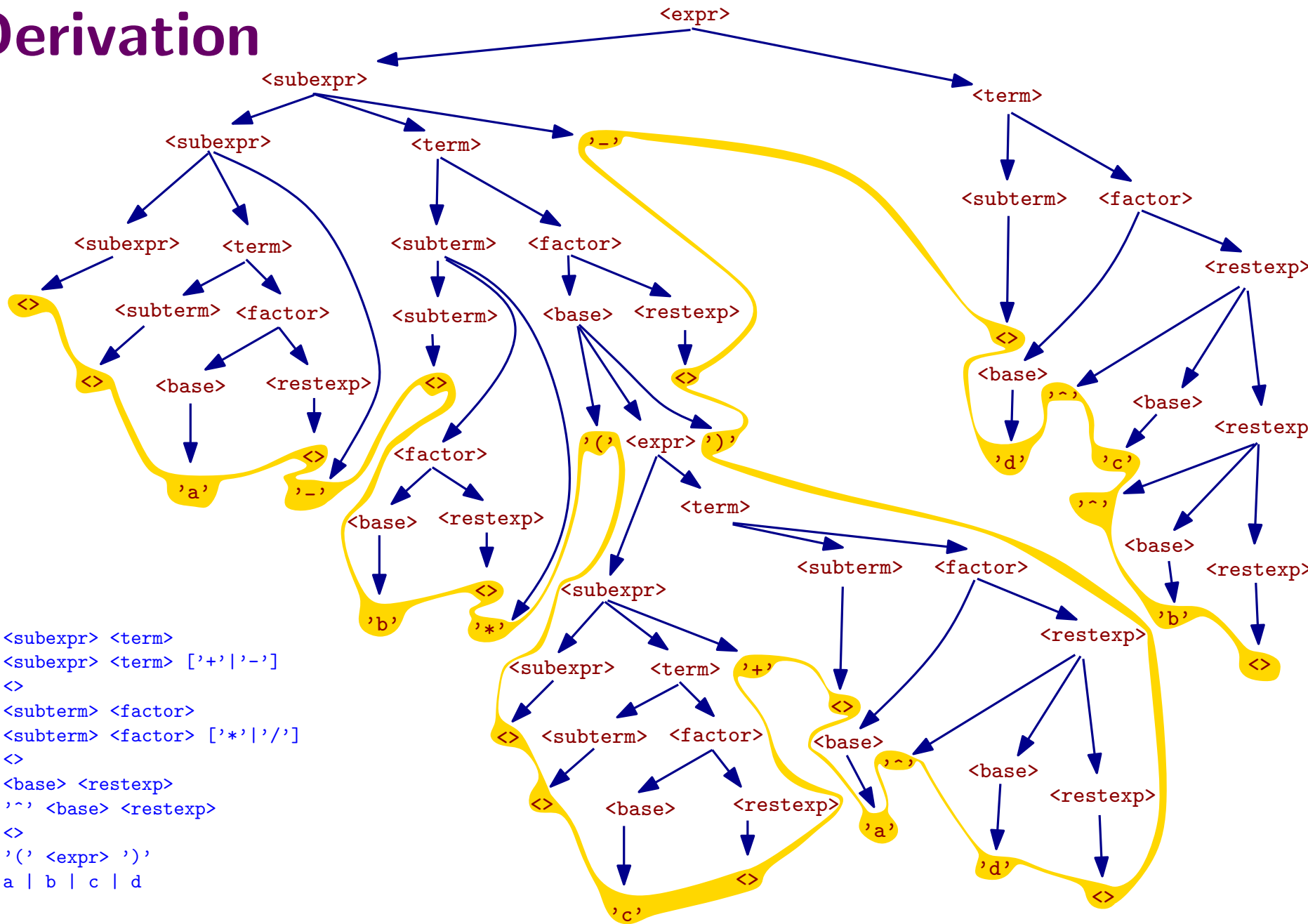
A Derivation




```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

a - b * (c + a ^ d) - d ^ c ^ b

A Derivation

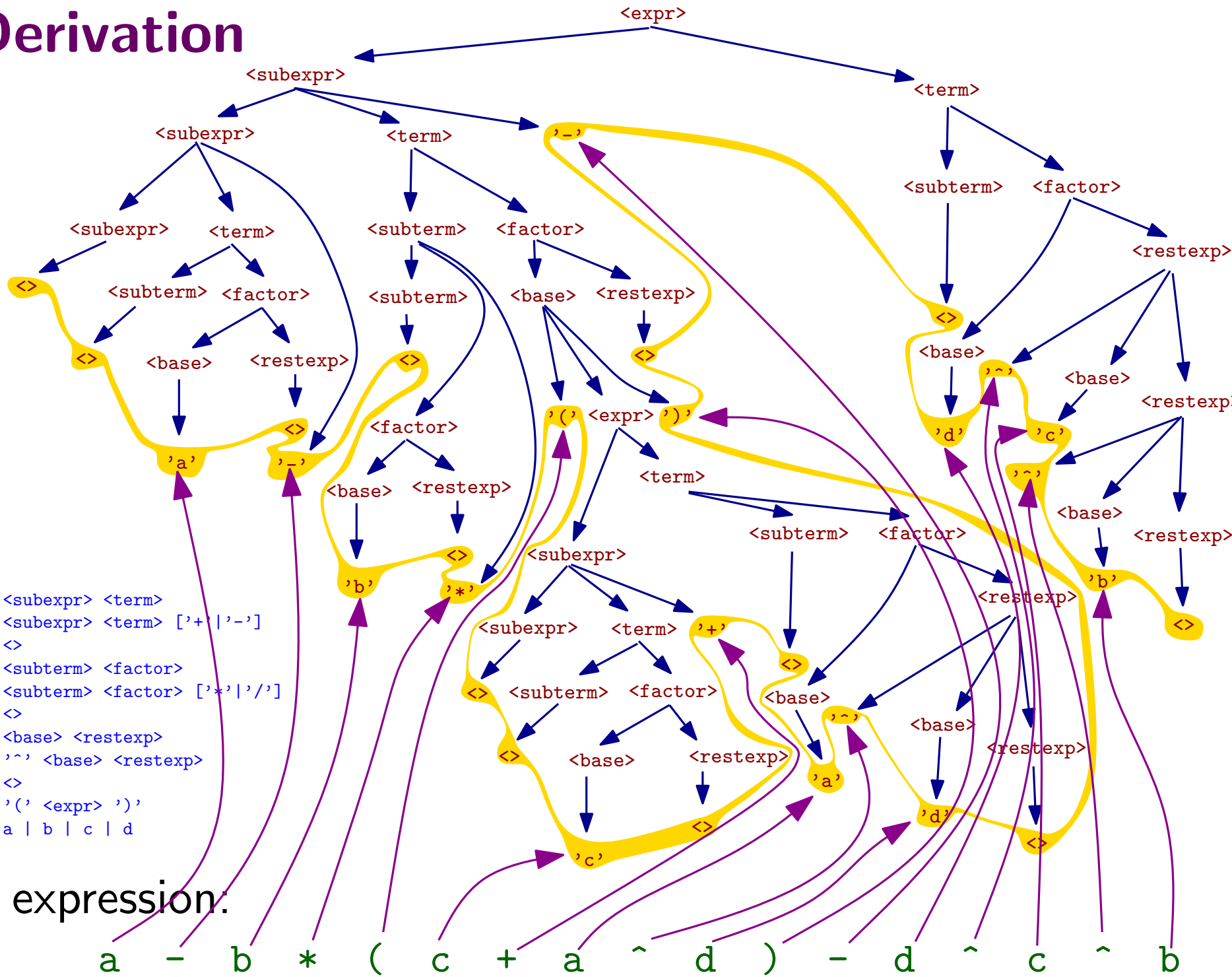



```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Parsed expression:

a - b * (c + a ^ d) - d ^ c ^ b

A Derivation

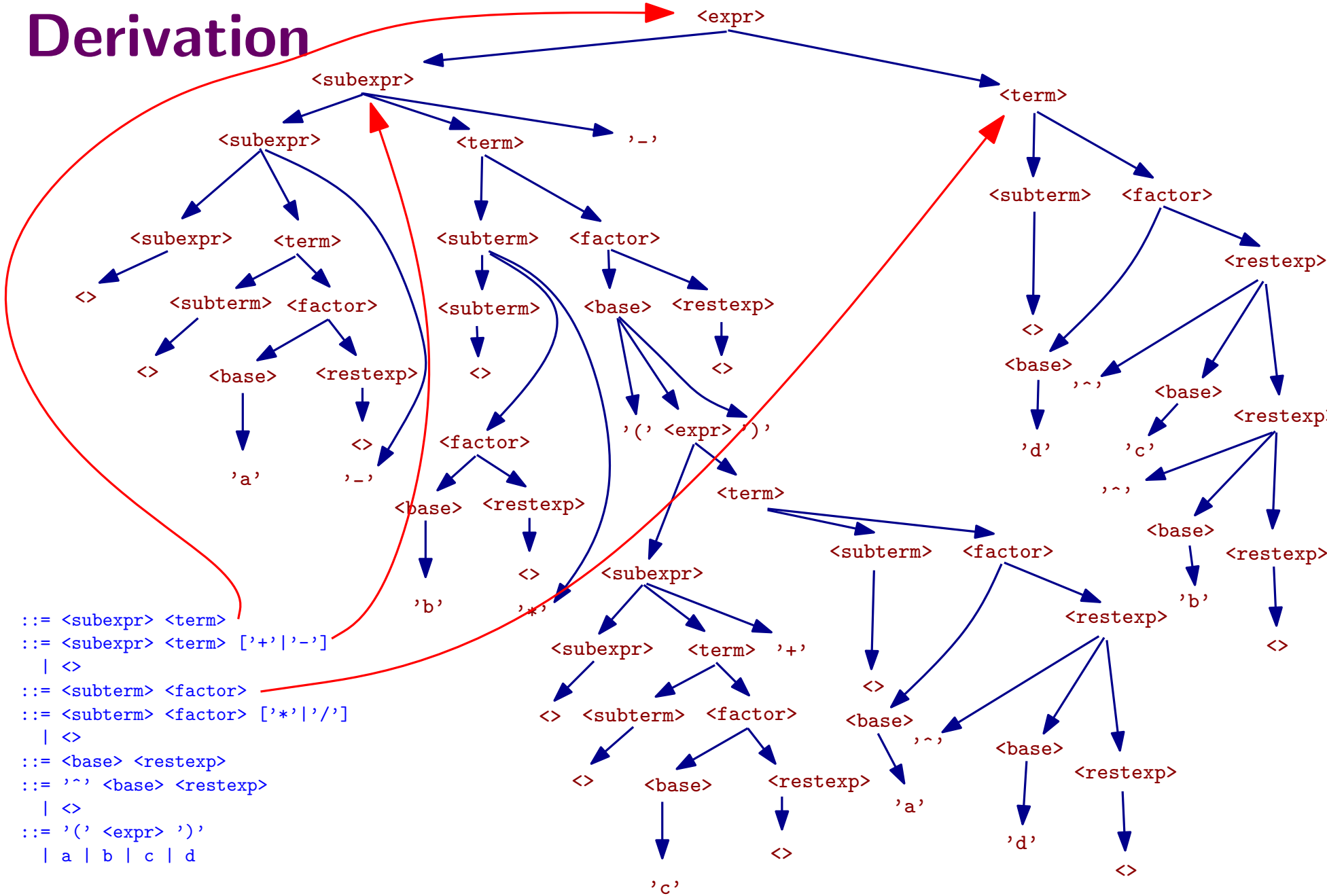



```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Parsed expression:

a - b * (c + a ^ d) - d ^ c ^ b

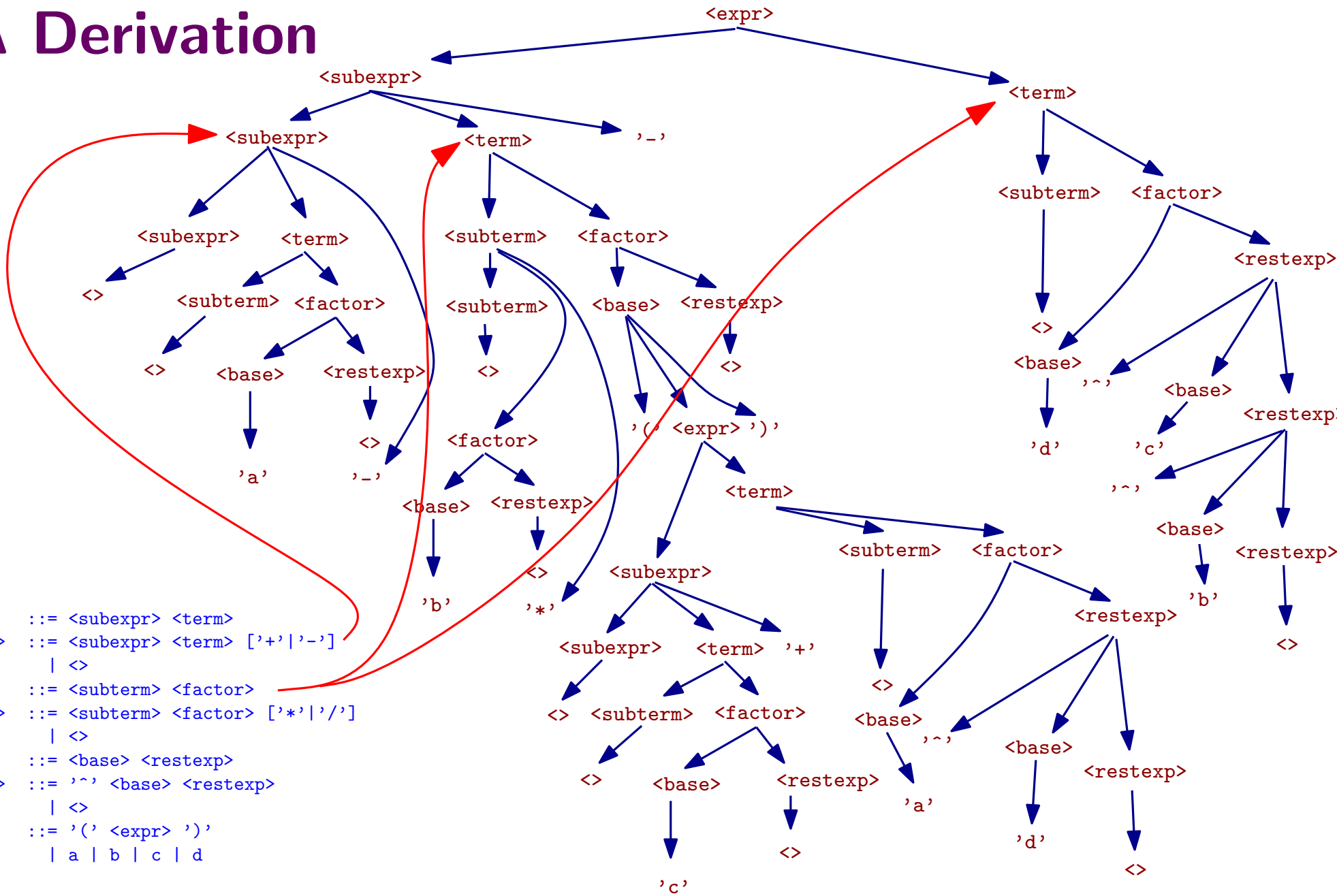
A Derivation




```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

a - b * (c + a ^ d) - d ^ c ^ b

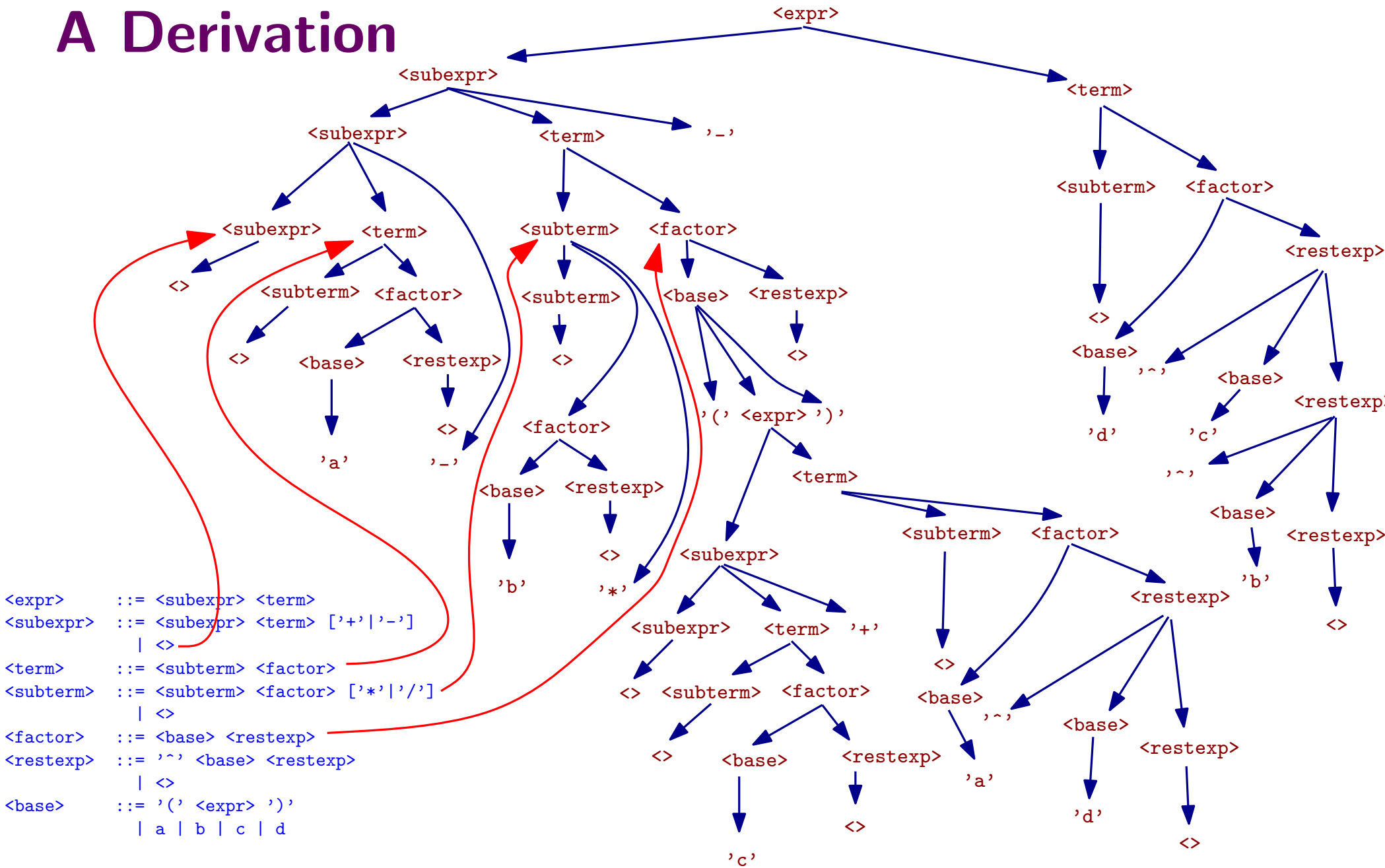
A Derivation




```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

a - b * (c + a ^ d) - d ^ c ^ b

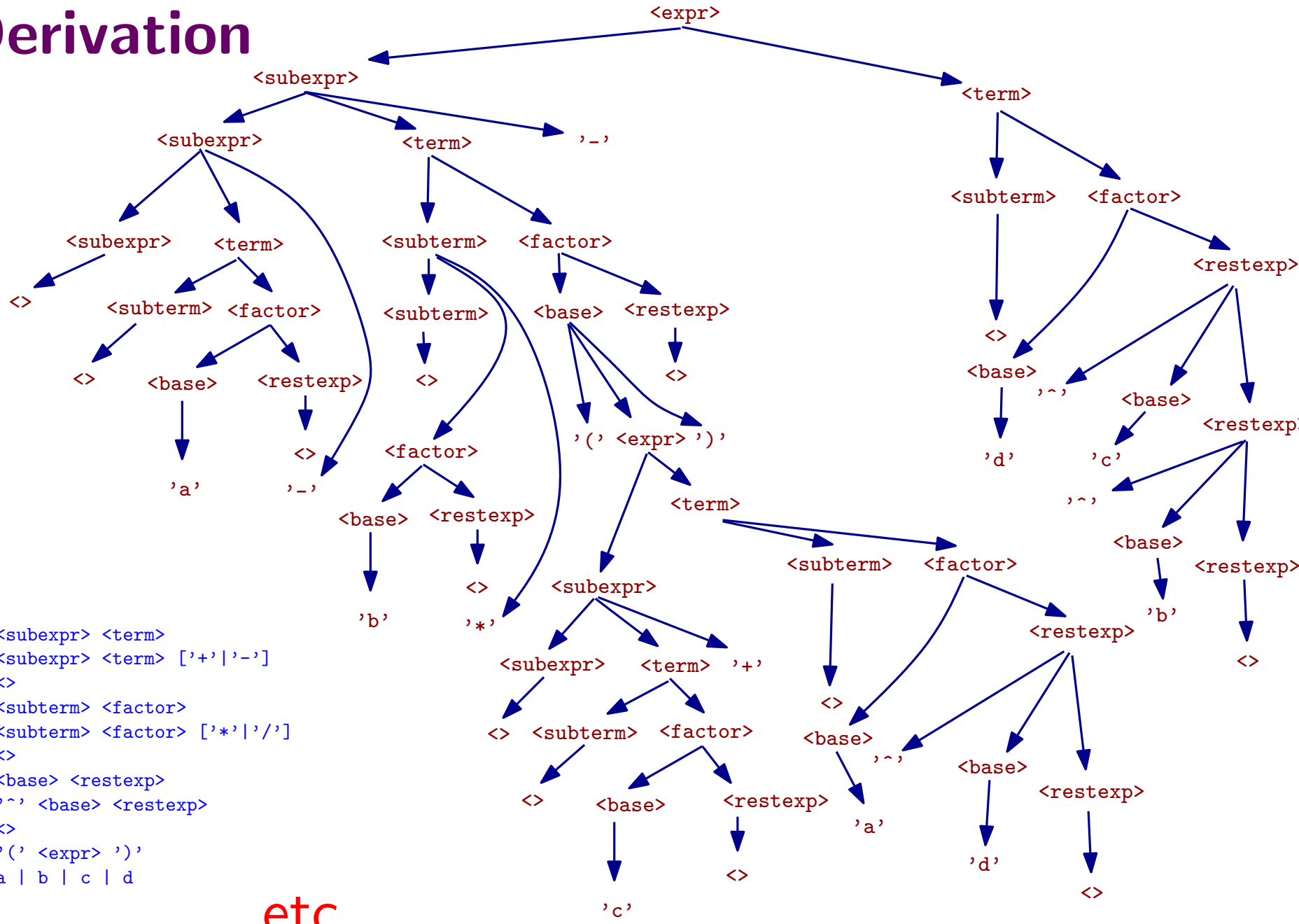
A Derivation




```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

a - b * (c + a ^ d) - d ^ c ^ b

A Derivation

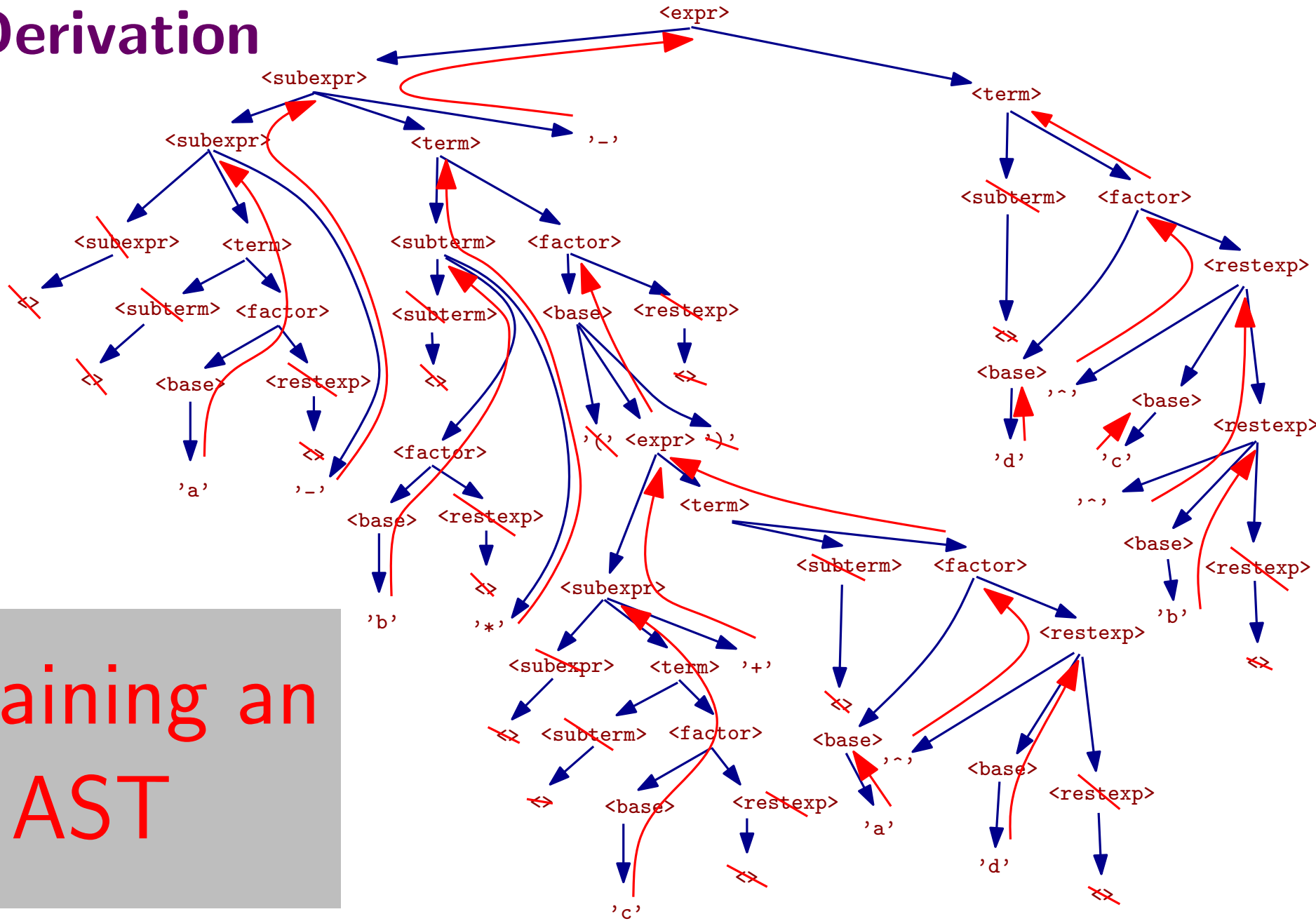



```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

etc...

a - b * (c + a ^ d) - d ^ c ^ b

A Derivation

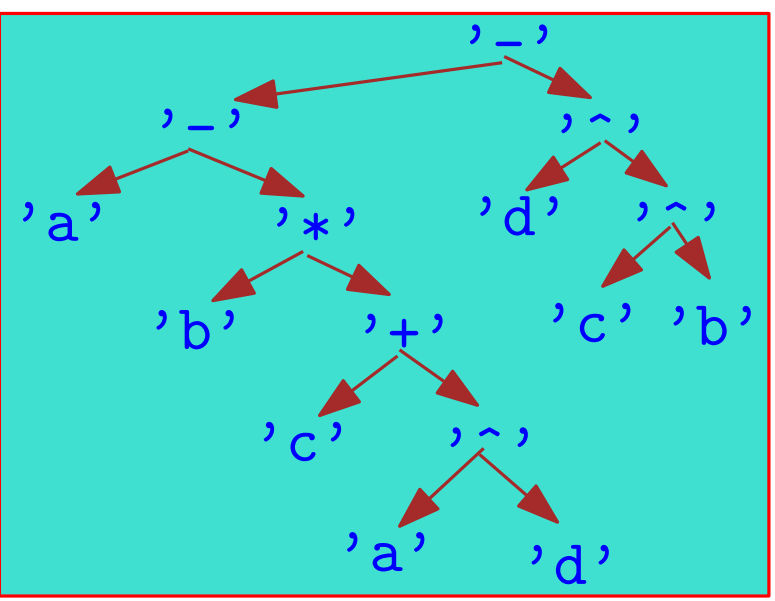


Obtaining an AST

a - b * (c + a ^ d) - d ^ c ^ b

A Derivation

AST:



a - b * (c + a ^ d) - d ^ c ^ b

Lessons Learned

- There are many grammars for the same language.
- *Parsing:* Building an AST for a given string, provided it is in the grammar's language.
- Unambiguous grammars are preferred for that purpose.
- Grammar should try to capture structural properties of the language, such as associativity of operators, and nesting of blocks.
- The AST can be built from the parse tree through an algorithmic process.
- Each tree segment is processed in a systematic way.

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal **<subexpr>**

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

Then string s is generated by nonterminal <expr>

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

Then string s is generated by nonterminal <expr>

Where s is s_1 concatenated with s_2.

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

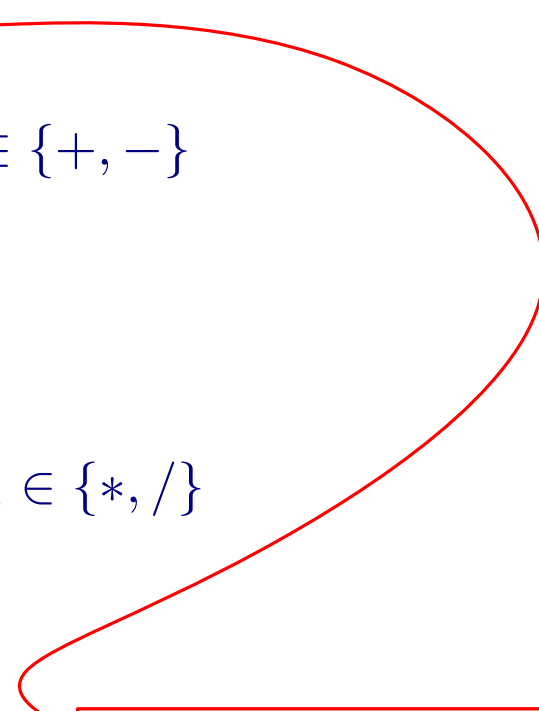
$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$



```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \widehat{\ } s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

etc...

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Prolog double quoted term: list of ASCII codes:

```
"abc" = [97,98,99]
```

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

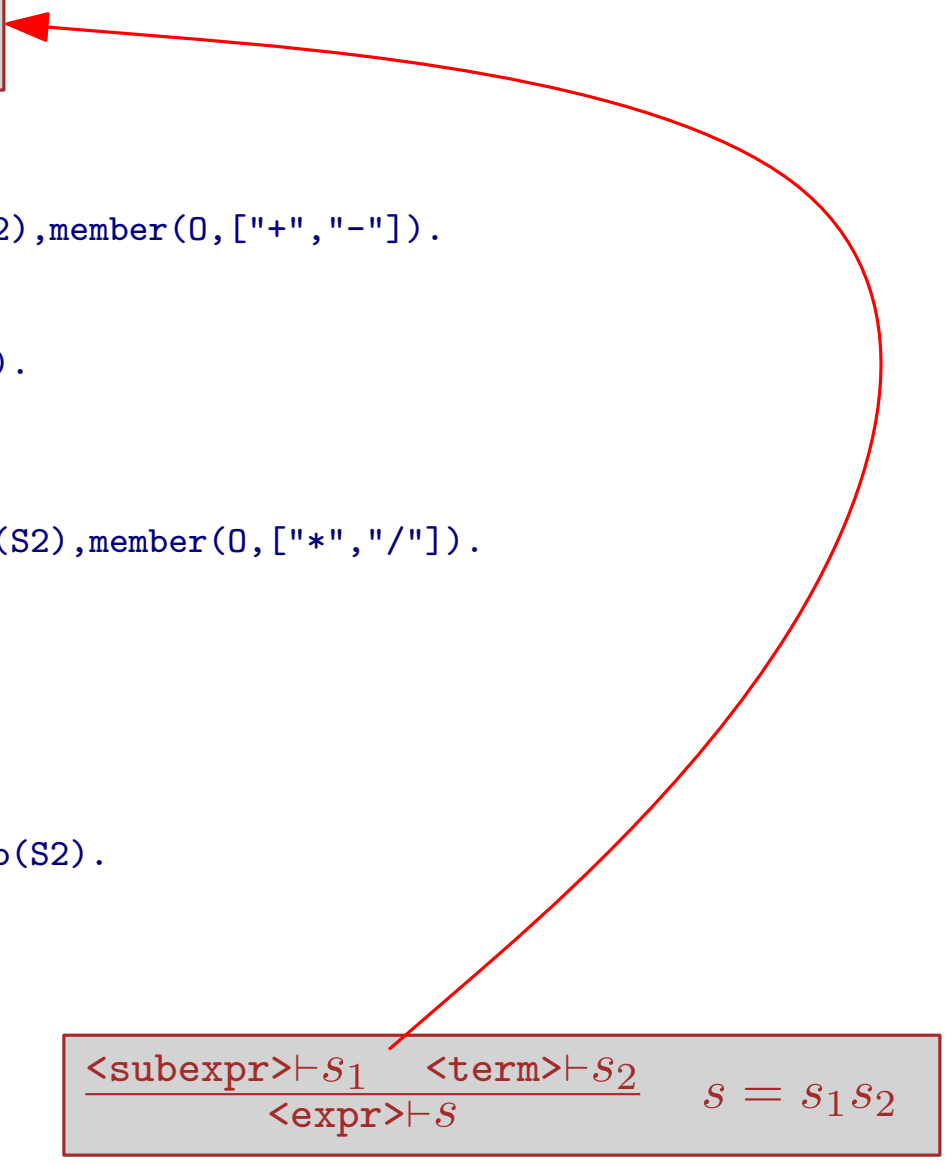
List of lists

Prolog double quoted term: list of ASCII codes:

"abc" = [97,98,99]

Prolog Code — Attempt 1

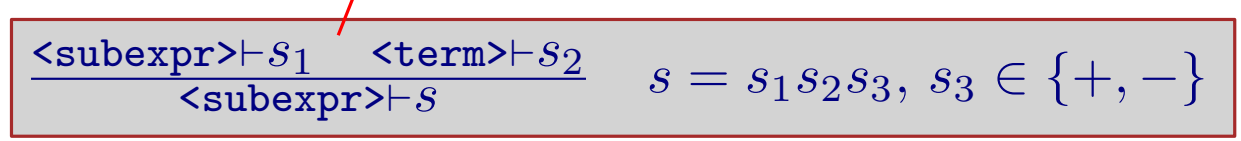
```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```



$$\frac{\texttt{<subexpr>} \vdash s_1 \quad \texttt{<term>} \vdash s_2}{\texttt{<expr>} \vdash s} \qquad s = s_1 s_2$$

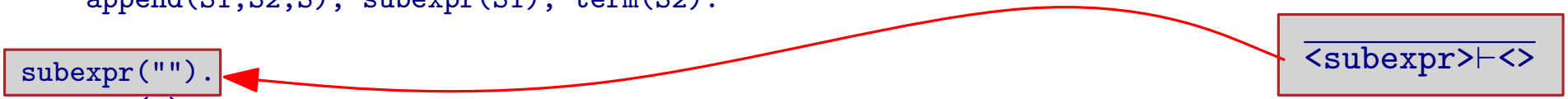
Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```



Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```



Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Empty string is prefix of empty string.

Left recursion

Runs into infinite loop !!!

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Add heuristics!

Non-empty
Balanced brackets
No + or - outside bracktes

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Add heuristics!

Non-empty
Balanced brackets
No + or - outside
bracktes

Non-empty
Balanced brackets
No * or / outside
bracktes

Non-empty
Balanced brackets
No ^ outside
bracktes

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

S2 is non-empty

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

S is concatenation of S1 and S2

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

No + or - outside brackets in S2

Characters outside brackets

```
constrain(S,S1,0,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   0 \= [] -> member(0,OL) ; true ).
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Watch demo on `findall` and `append/2`

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

S2 should contain balanced brackets only

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

S is S1 concatenated with S2 and O, where O is either + or -

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Apply heuristics in all places where it is useful!

Prolog Code — Attempt 2

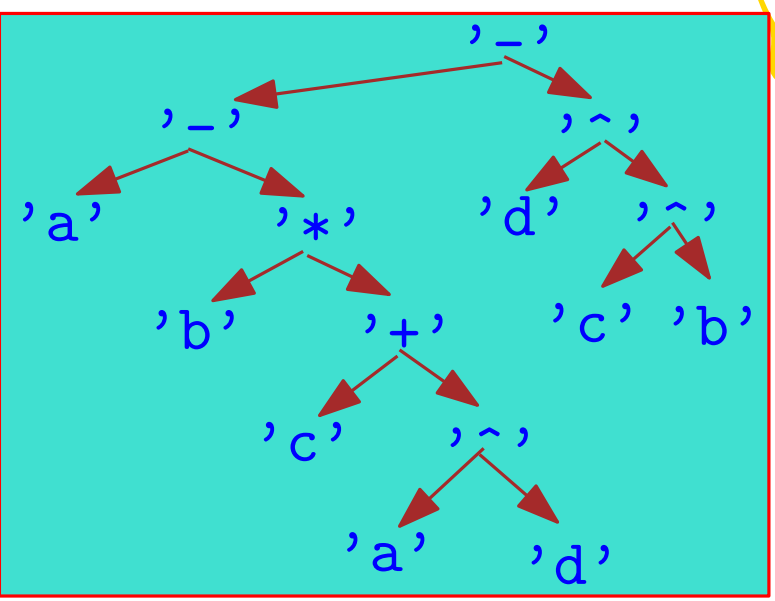
```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Query:

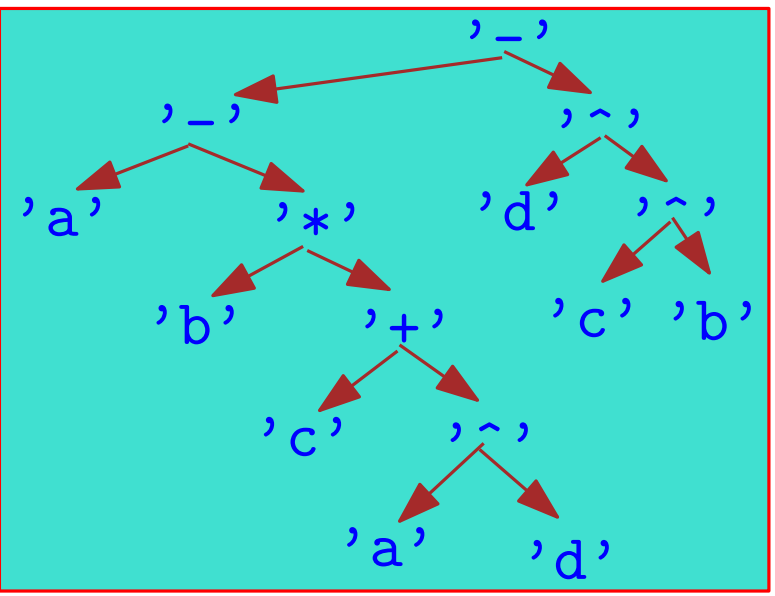
```
1 ?- S="(((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b)", expr(S).
S = [40, 40, 40, 97, 43, 98, 41, 42, 99|...].
```

AST



a - b * (c + a ^ d) - d ^ c ^ b

AST



Residual operators need to be hoisted 2 levels up.

Residual operators

a - b * (c + a ^ d) - d ^ c ^ b

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Second argument is where the AST is output

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Residual operator

Third argument of some nonterminals

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Convert ASCII code S into atom A.

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).
```

```
build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Piggyback on the syntax analyzer

Tree building:

- If the residual operator is `nil`, just pass the current tree up.
- If the residual operator is not nil, then `L` contains the tree components, which must be assembled into a term.

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Query:

```
1 ?- S="(((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b)",
     expr(S,T), T =.. L, S =.. X.
S = [40, 40, 40, 97, 43, 98, 41, 42, 99|...],
T = (((a+b)*c/d^e^f-g)^ (a*b)+c)* (a+b),
L = [*, ((a+b)*c/d^e^f-g)^ (a*b)+c, a+b],
X = ['.', 40, [40, 40, 97, 43, 98, 41|...]].
```

YACC : separate set of slides