Languages, Grammars, Regular Expressions

Synopsis

- **Languages:** how to define
- **Grammars**
  - Specification of languages
  - Language generators
  - Derivations
  - Analysis
  - Language Structure
- **Regular Languages**
  - Regular grammars
  - Deterministic Finite Automata
  - Regular expressions
  - Use of REs in Ruby

Languages

- Symbols: elements of an alphabet, typically ASCII symbols
- Strings: sequences of symbols.
- Language: *set of strings*

Examples of languages:

$$\{a, ab, aba, bab, bbb\}$$

$$\{0, 1, \ldots, 9, 10, 11, \ldots, 19, 21, \ldots\}$$

$$\left\{ \texttt{int main()\{printf("Hello world");\}} \; , \ldots \right\}$$

Languages

- Symbols: elements of an alphabet, typically ASCII symbols
- Strings: sequences of symbols.
- Language: *set of strings*

The language C is the set of all possible C programs!

Examples of languages:

$$\{a, ab, aba, bab, bbb\}$$

$$\{0, 1, \ldots, 9, 10, 11, \ldots, 19, 21, \ldots\}$$

$$\left\{ \texttt{int main()\{printf("Hello world");\}}\ , \ldots \right\}$$

Specification of Languages

- Enumeration is not an efficient way of specifying a language.
- Must be finitary.
- Must provide an efficient way of deciding whether a string belongs to the language or not.
- Must capture the structure of the language.
    - For PL, execution must take structure into account.
- **Grammars:** standard mechanism of specifying programming languages

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

Production rule

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

Production Rules

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

Set of production rules

Production Rules

Left hand side

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

Right hand side

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

Production Rules

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

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

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

Derivation:

$E \rightarrow$

Production Rules

Derivation:

$$\boxed{E} \rightarrow \boxed{E * E}$$

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

Production Rules

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

REWRITE

Derivation:

$$
\begin{aligned}
E &\rightarrow E * E \\
&\rightarrow (E) * E
\end{aligned}
$$

Production Rules

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

REWRITE

Derivation:

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

Production Rules

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

REWRITE

Derivation:

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

Production Rules

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

REWRITE

Derivation:

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

Production Rules

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

REWRITE

Derivation:

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

Production Rules

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

REWRITE

Derivation:

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

Production Rules

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

REWRITE

Derivation:

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

Production Rules

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

Derivation:

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

Production Rules

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

Derivation:

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

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

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:

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

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

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:

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

In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

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

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

In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

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

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

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

In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

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

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

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

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

In practice...

Rules are enough in practice.

Grammar can be derived from the rules:

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

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

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

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

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)

$$\boxed{G = \langle \Sigma, N, \Pi, E \rangle}$$

Derivation Tree

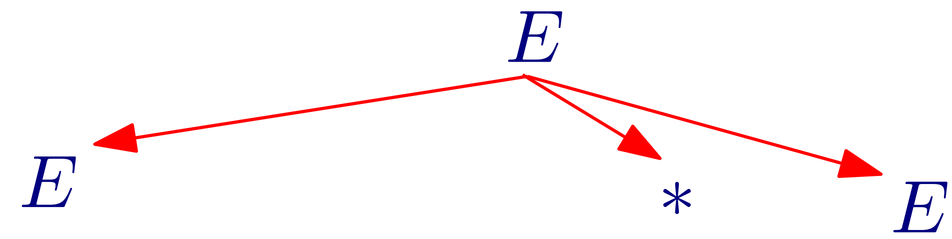
$$
\begin{aligned}
E &\rightarrow (E) \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow D \\
D &\rightarrow 0D \\
D &\rightarrow 1D \\
D &\rightarrow 0 \\
D &\rightarrow 1
\end{aligned}
$$

E

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

Derivation Tree

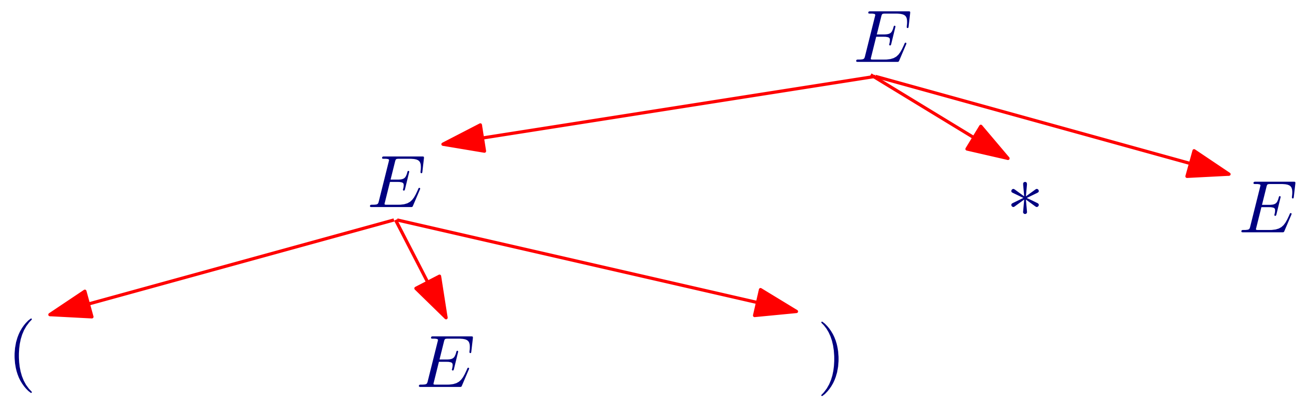
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



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

Derivation Tree

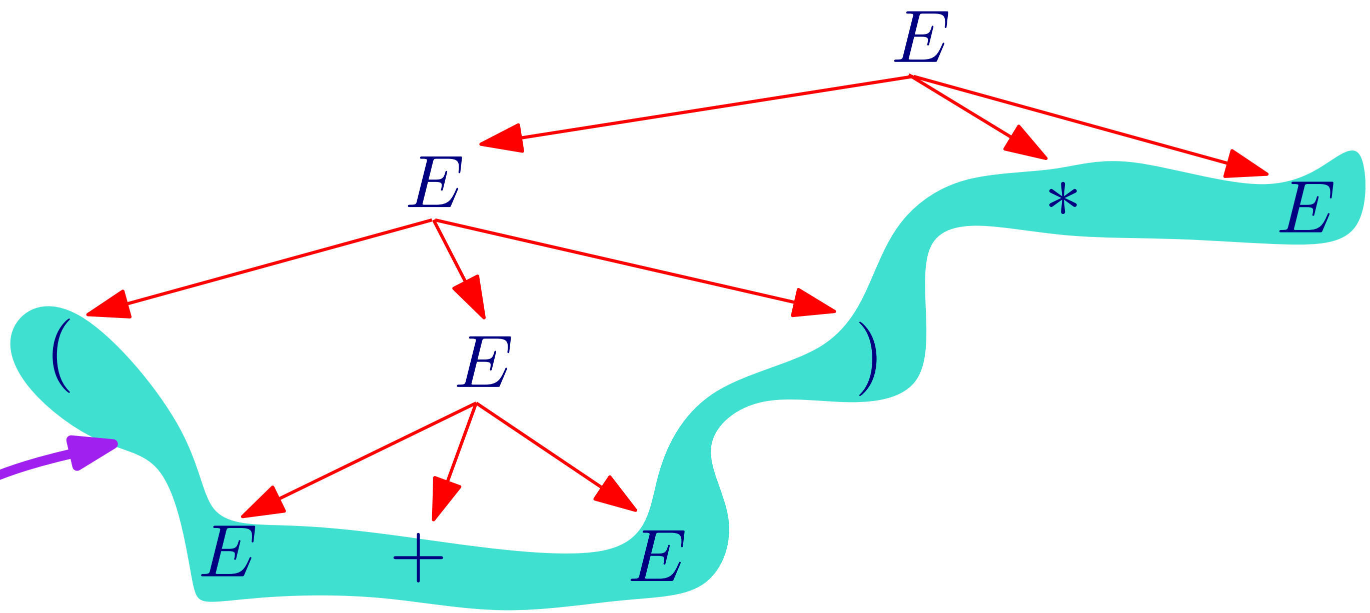
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



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

Derivation Tree

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

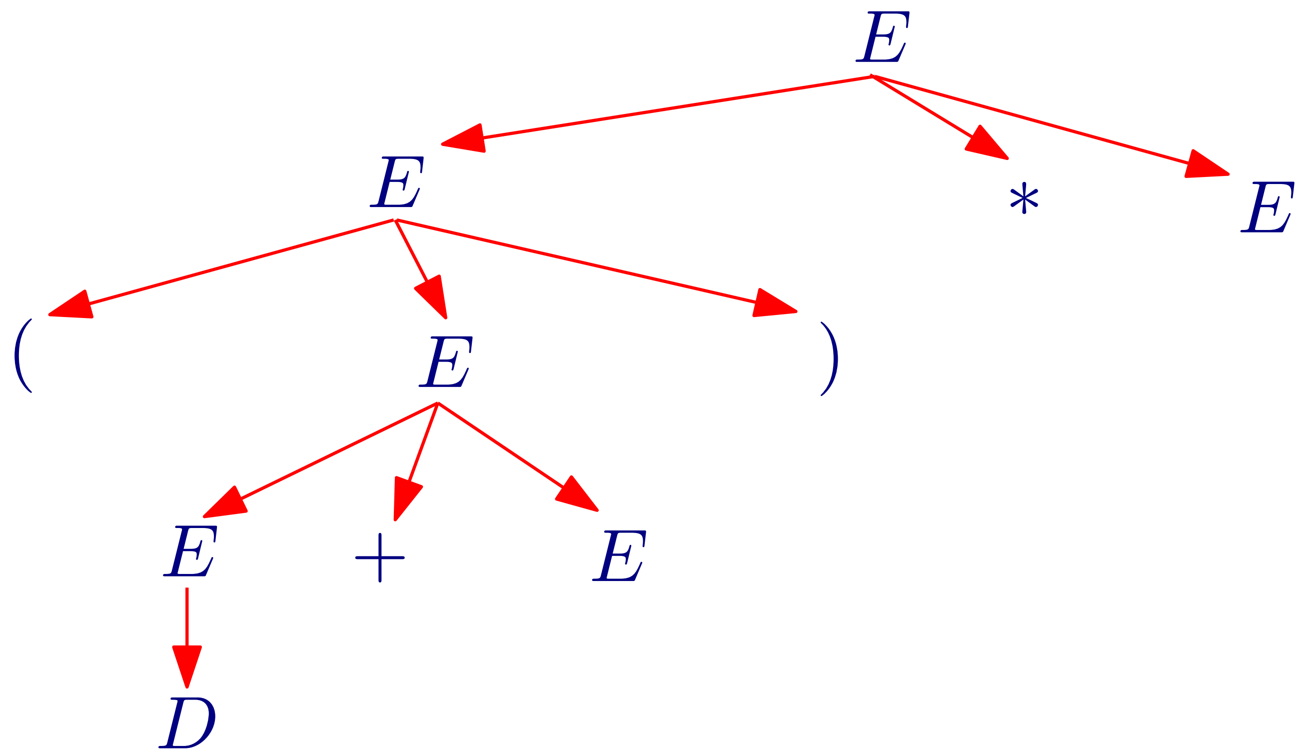


Current frontier of derivation tree: *sentential form*

matched: $($ 1 $+$ 1 0 $)$ $*$ 1

Derivation Tree

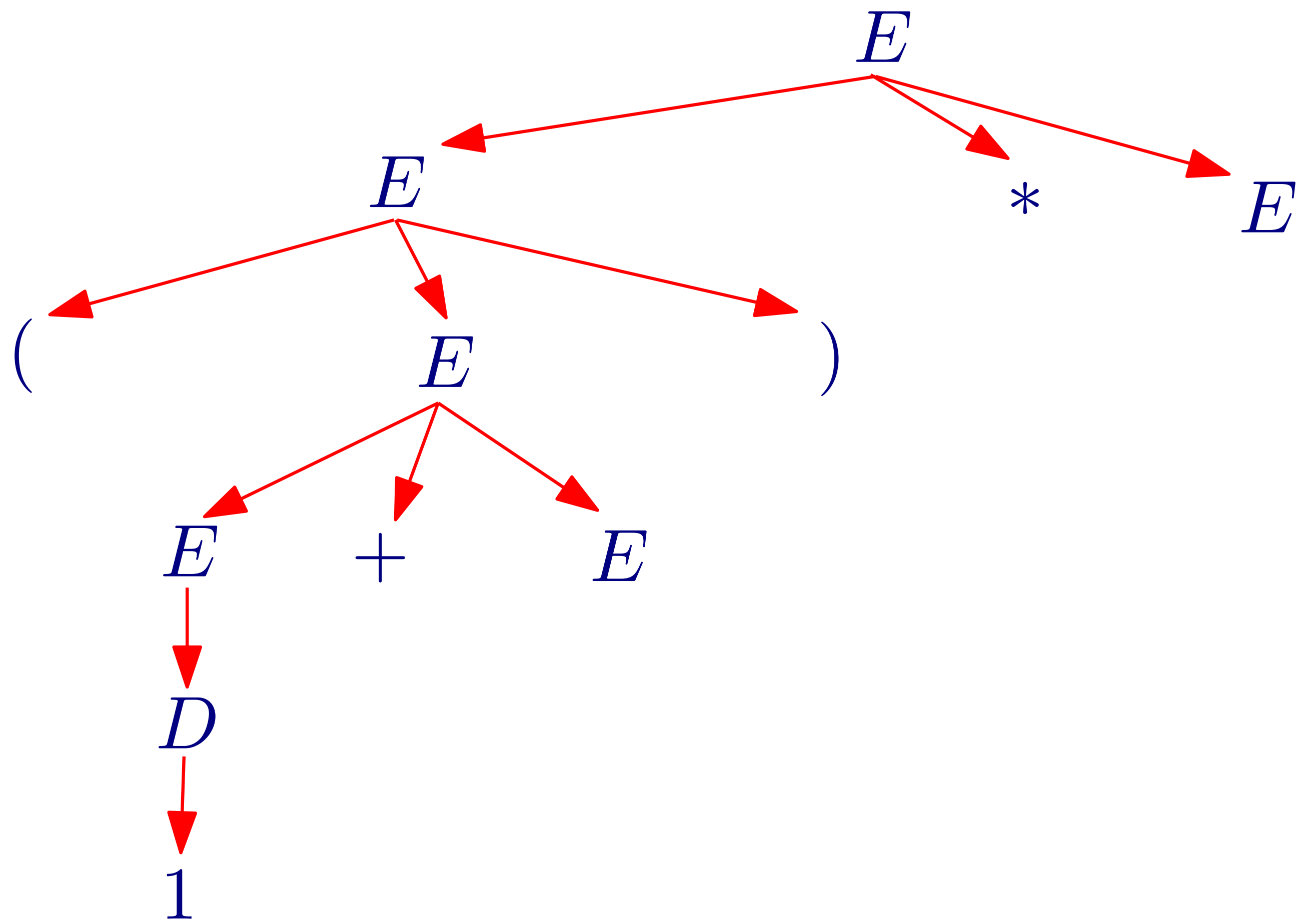
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



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

Derivation Tree

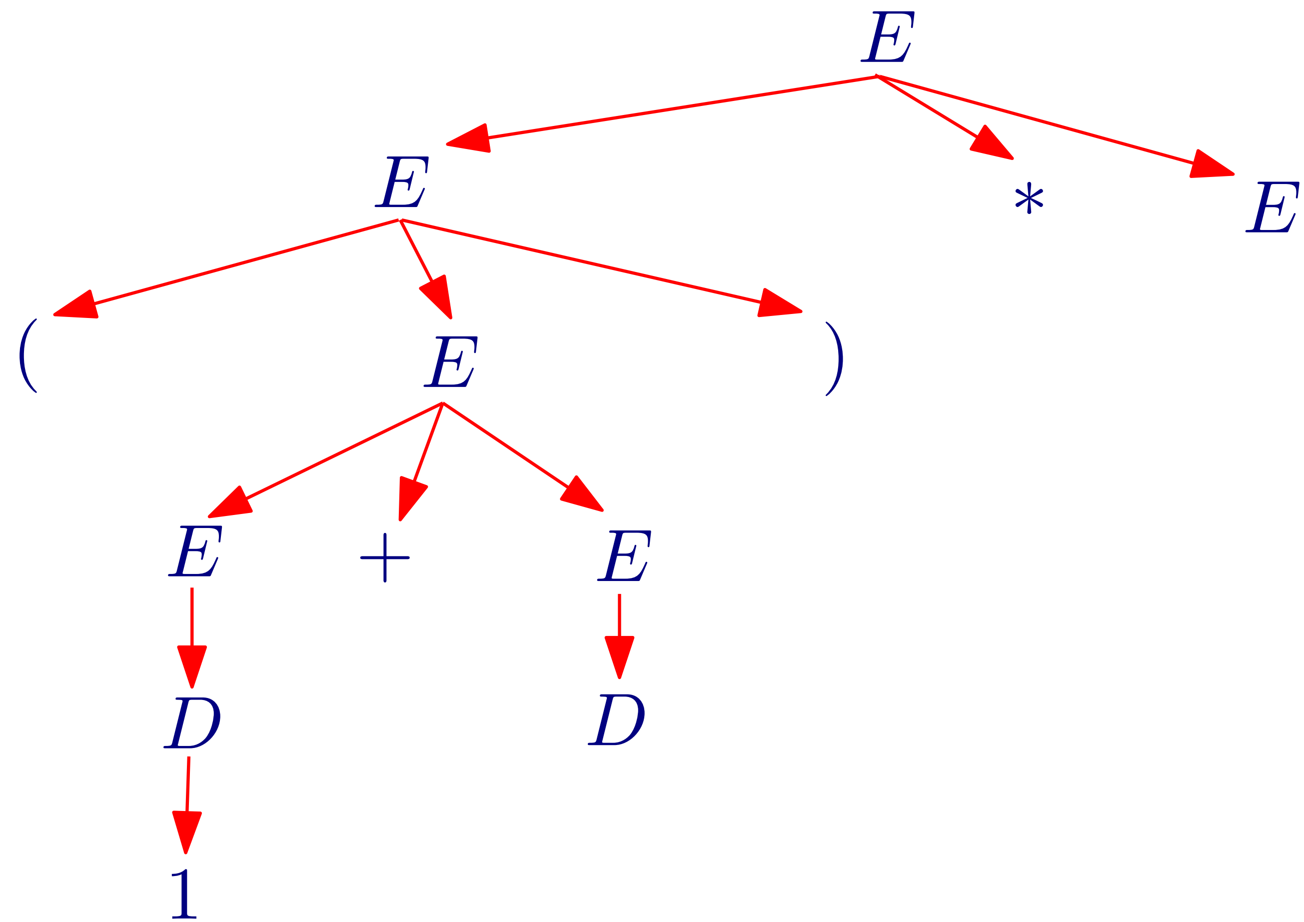
$$\begin{array}{lcl} E & \rightarrow & (E) \\ E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & D \\ D & \rightarrow & 0D \\ D & \rightarrow & 1D \\ D & \rightarrow & 0 \\ D & \rightarrow & 1 \end{array}$$



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

Derivation Tree

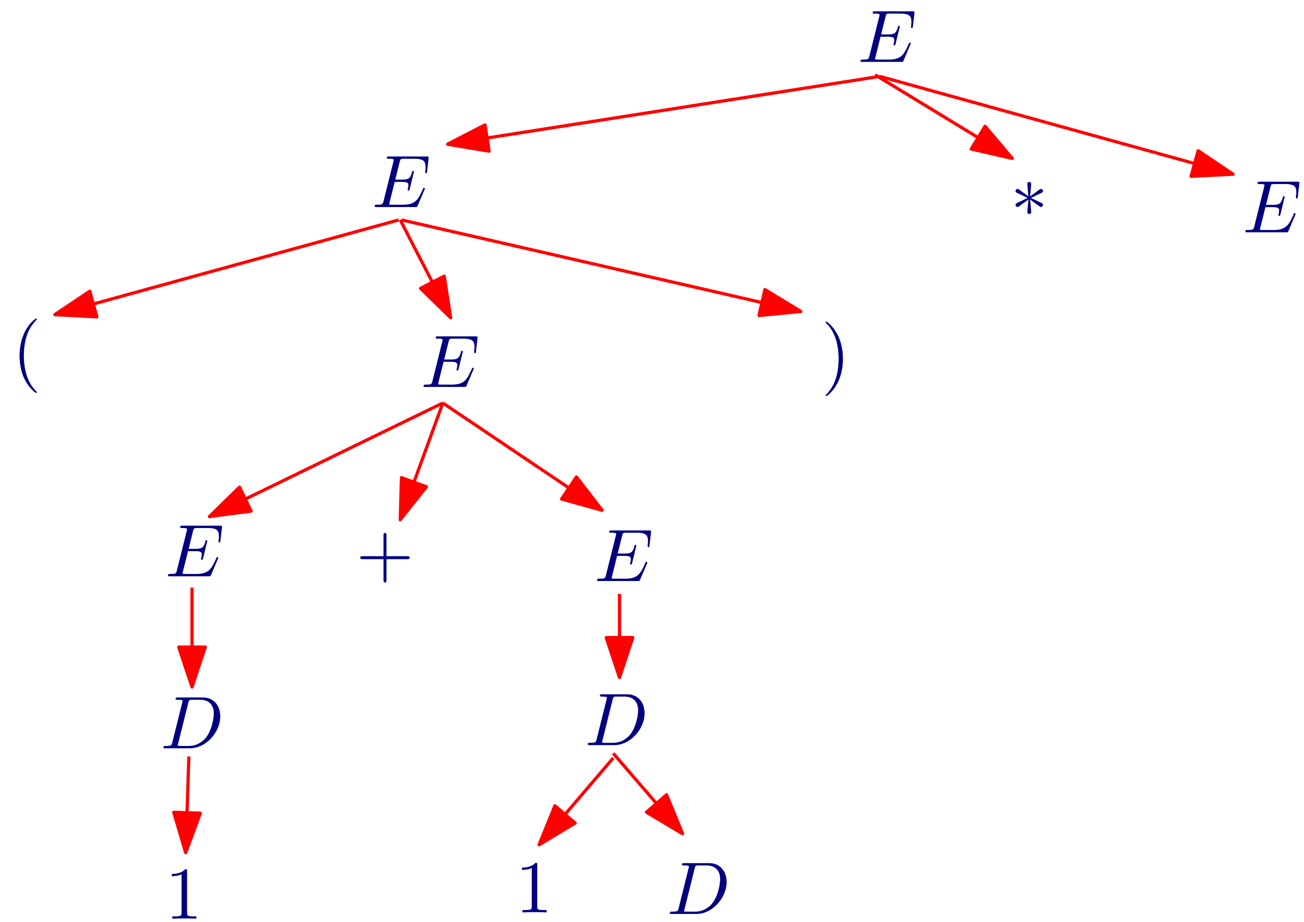
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$



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

Derivation Tree

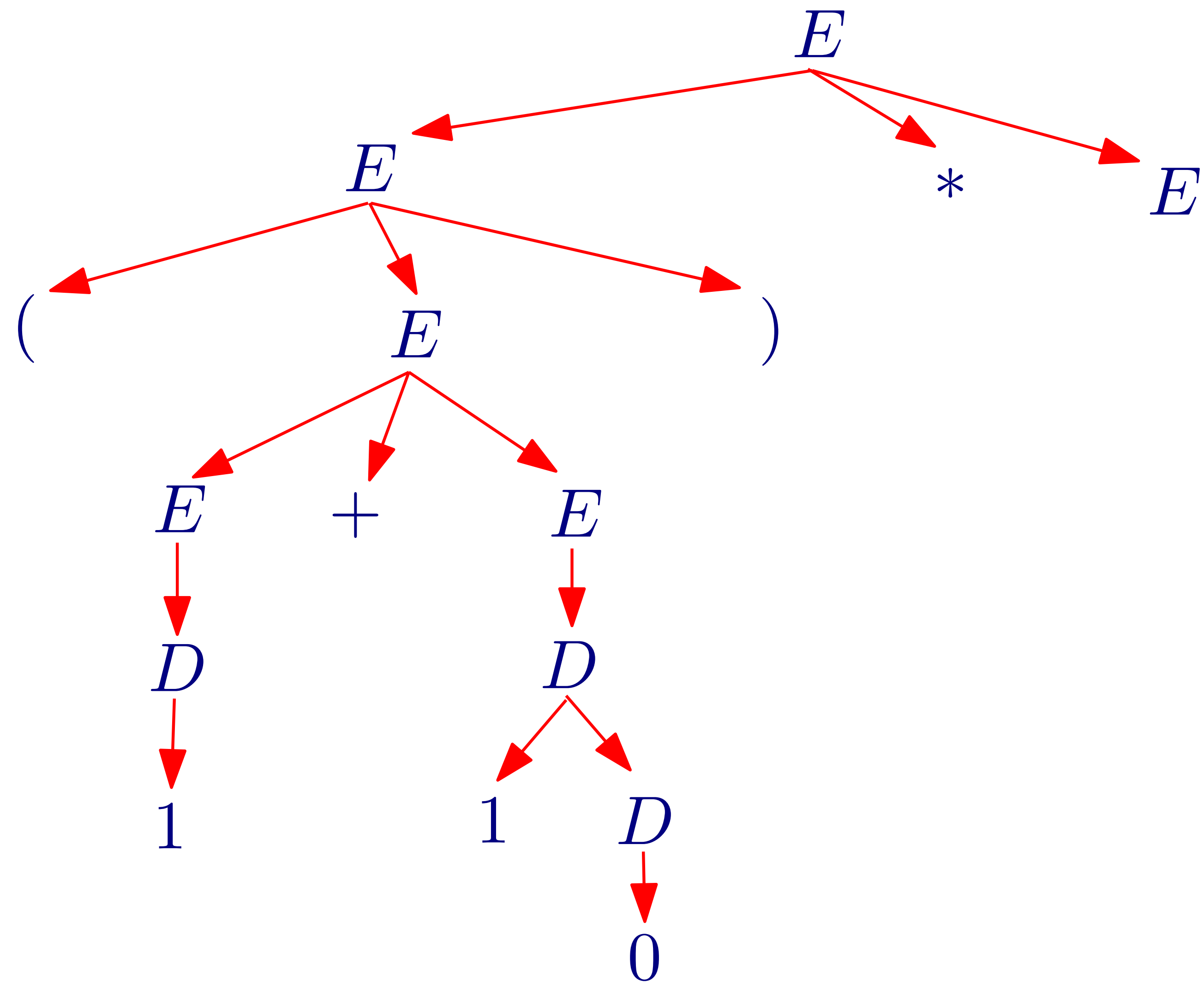
$$\begin{array}{lcl} E & \rightarrow & (E) \\ E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & D \\ D & \rightarrow & 0D \\ D & \rightarrow & 1D \\ D & \rightarrow & 0 \\ D & \rightarrow & 1 \end{array}$$



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

Derivation Tree

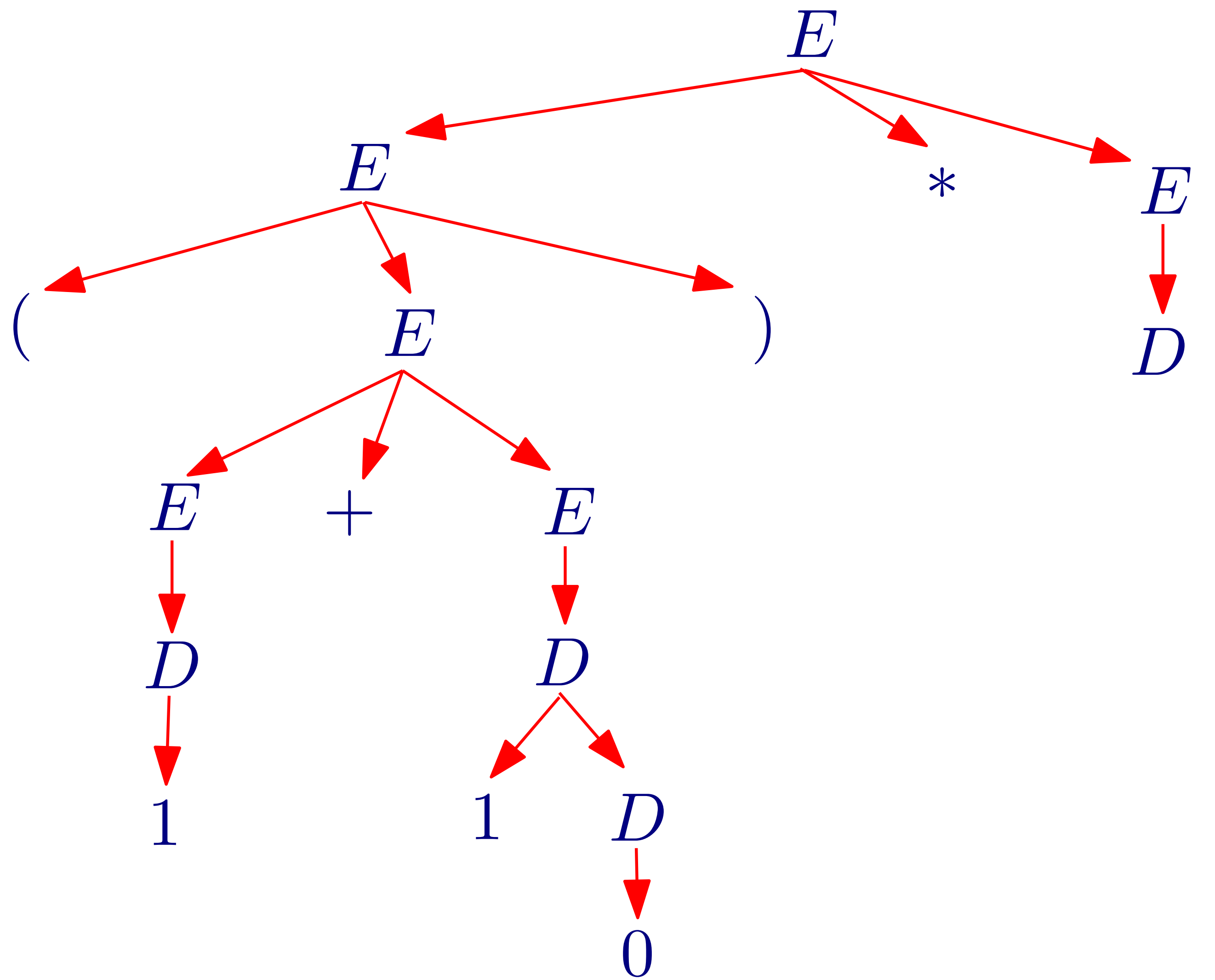
$$
\begin{aligned}
E &\rightarrow (E) \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow D \\
D &\rightarrow 0D \\
D &\rightarrow 1D \\
D &\rightarrow 0 \\
D &\rightarrow 1
\end{aligned}
$$



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

Derivation Tree

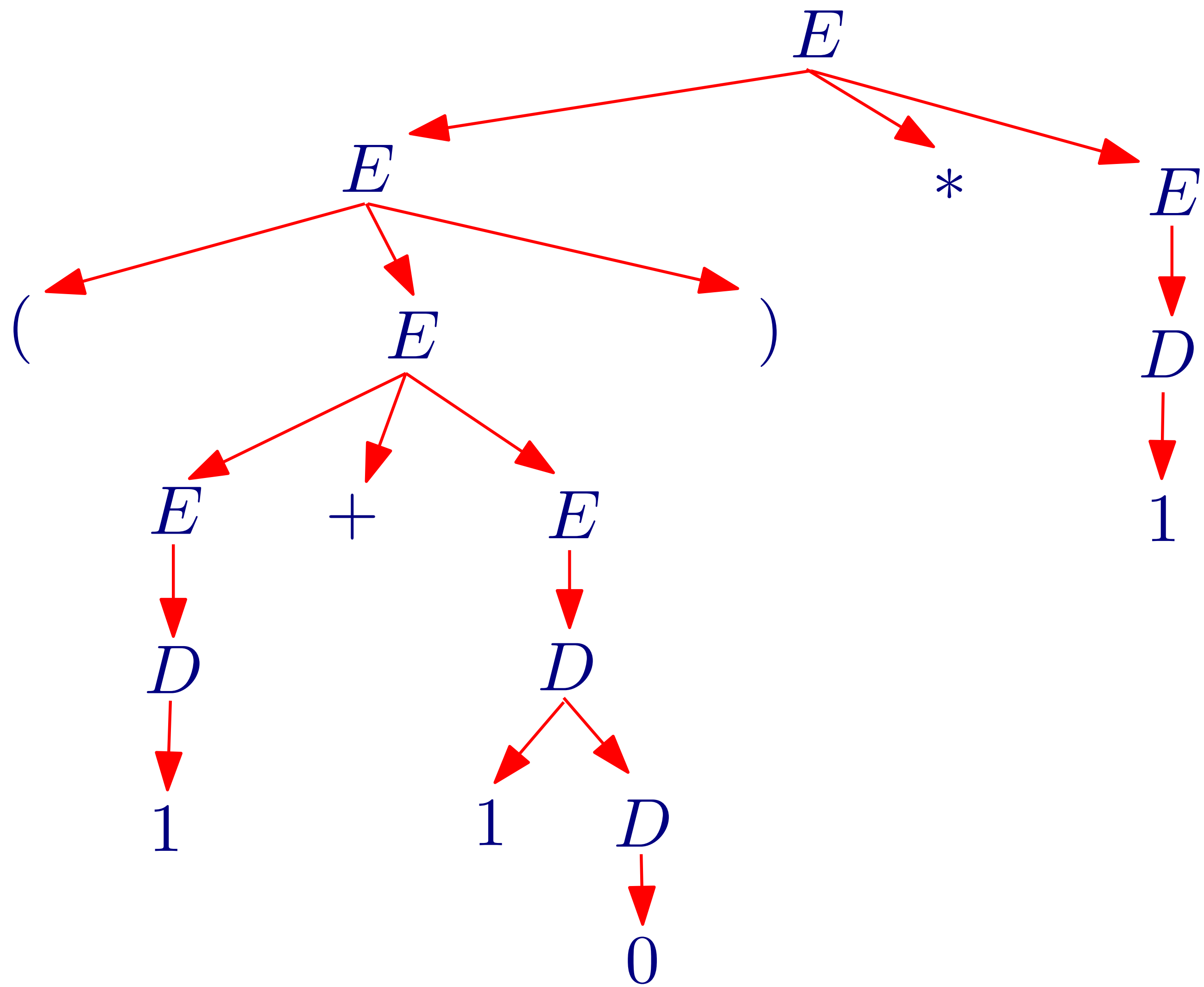
$$\begin{array}{lcl} E & \rightarrow & (E) \\ E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & D \\ D & \rightarrow & 0D \\ D & \rightarrow & 1D \\ D & \rightarrow & 0 \\ D & \rightarrow & 1 \end{array}$$



matched: (1 + 1 0) * 1

Derivation Tree

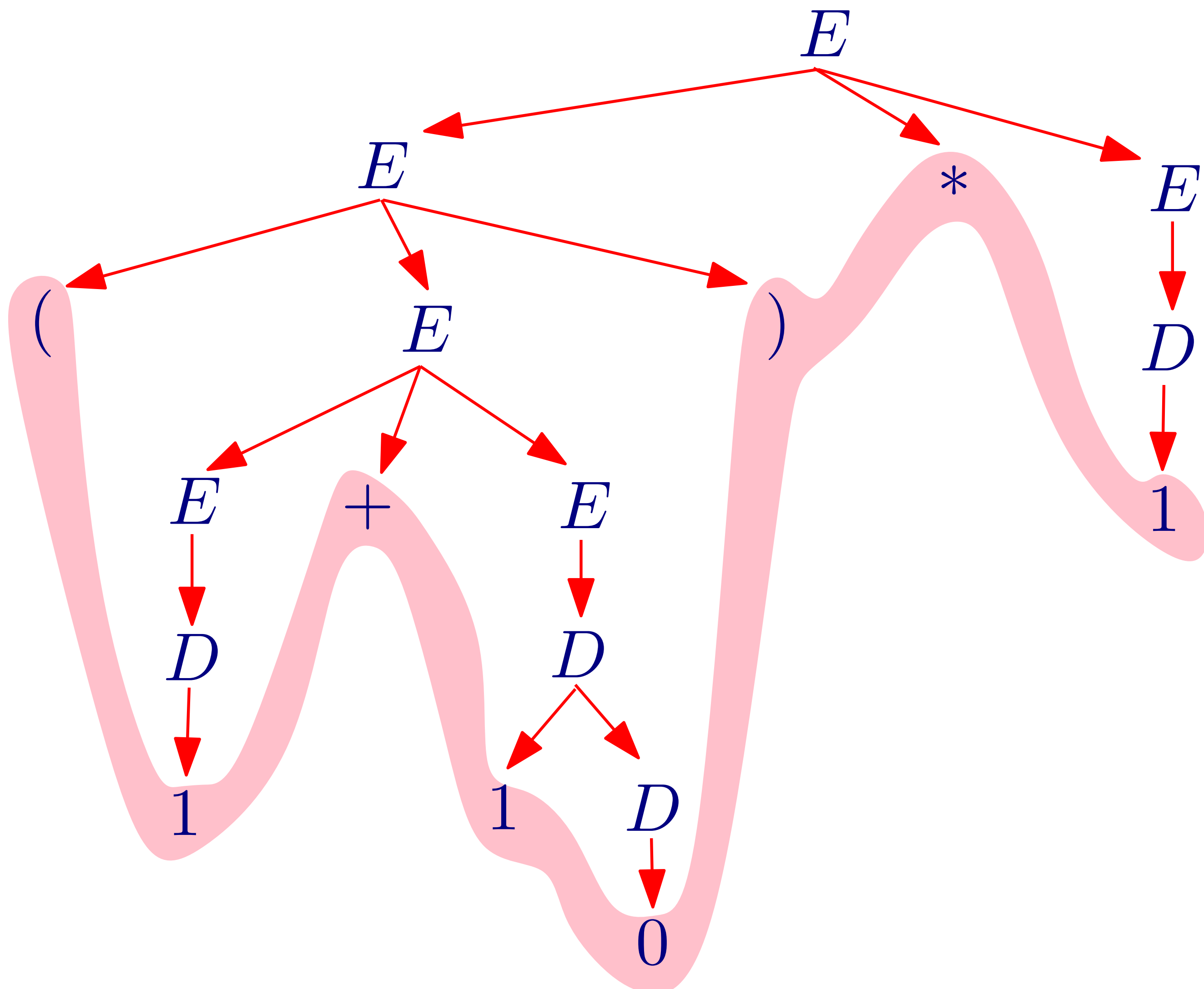
$$
\begin{aligned}
E &\rightarrow (E) \\
E &\rightarrow E + E \\
E &\rightarrow E * E \\
E &\rightarrow D \\
D &\rightarrow 0D \\
D &\rightarrow 1D \\
D &\rightarrow 0 \\
D &\rightarrow 1
\end{aligned}
$$



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

Derivation Tree

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

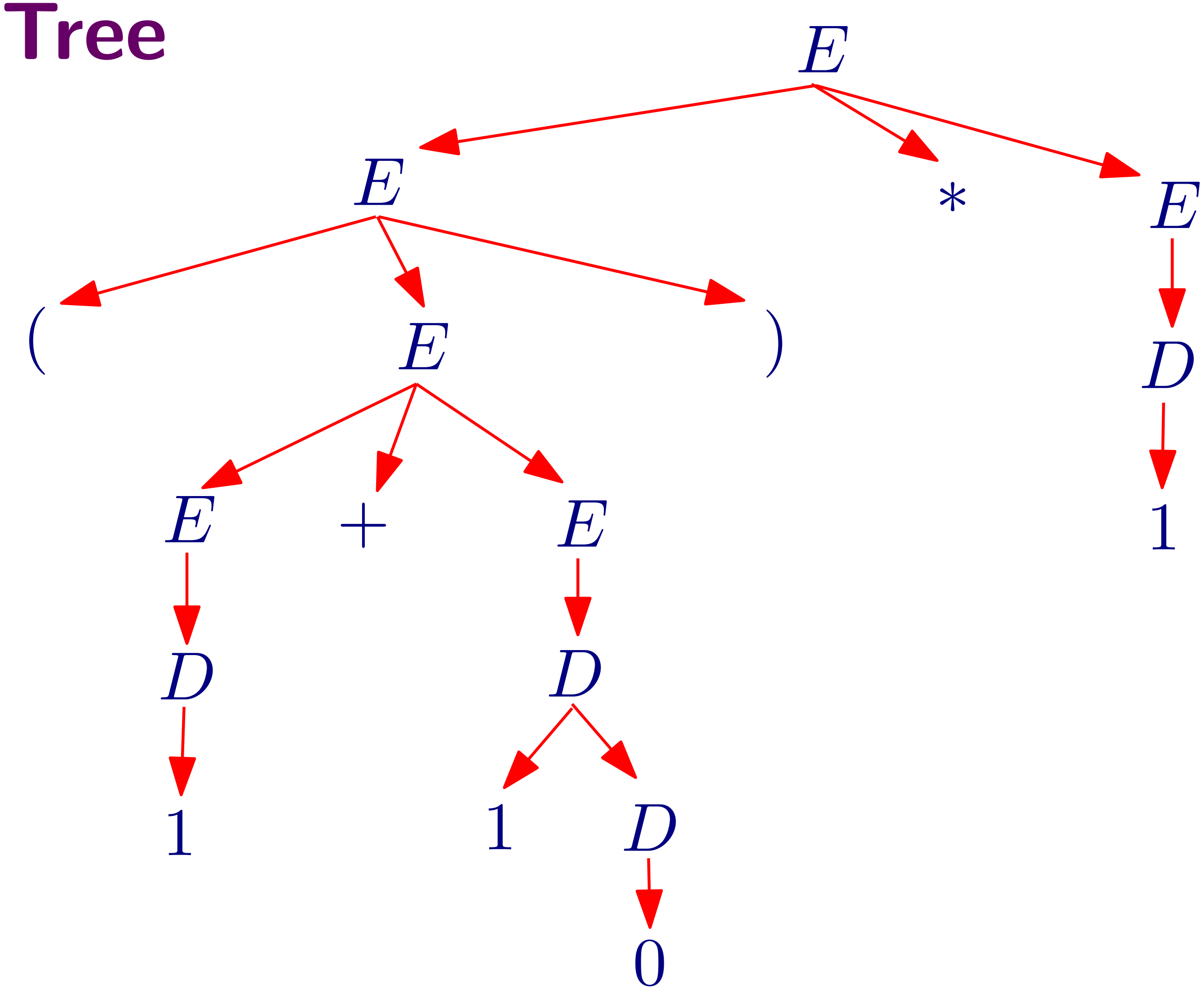


matched: $(\quad 1 \quad + \quad 1 \quad 0 \quad) \quad * \quad 1$

Abstract Syntax Tree

Simplified version of the parse tree:

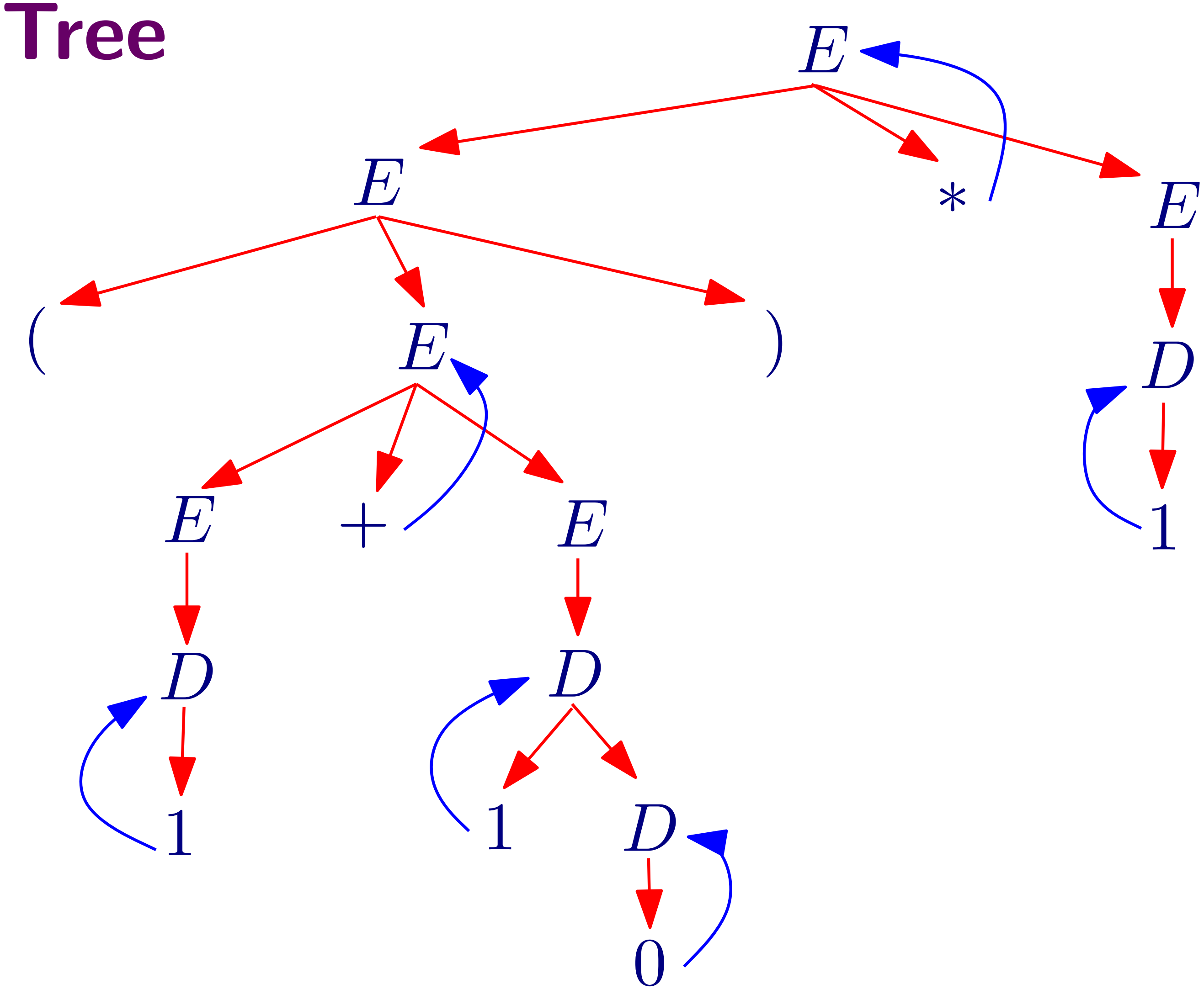
- ◇ if node has only one terminal child, make the terminal the label of current node and remove the child
- ◇ if node has only one child, shortcut the node
- ◇ many other customized rules, discussed as they are needed



Abstract Syntax Tree

Simplified version of the parse tree:

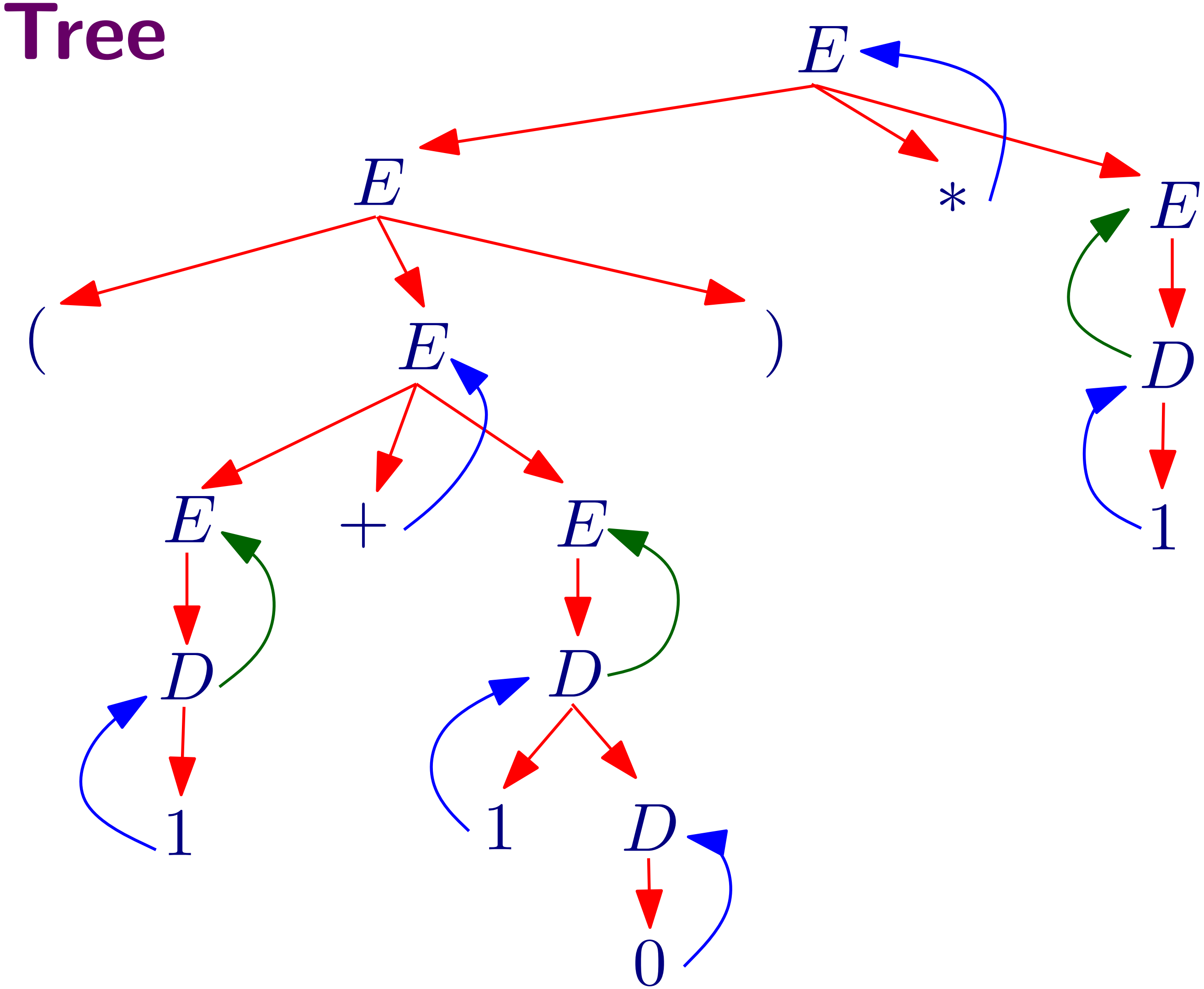
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Abstract Syntax Tree

Simplified version of the parse tree:

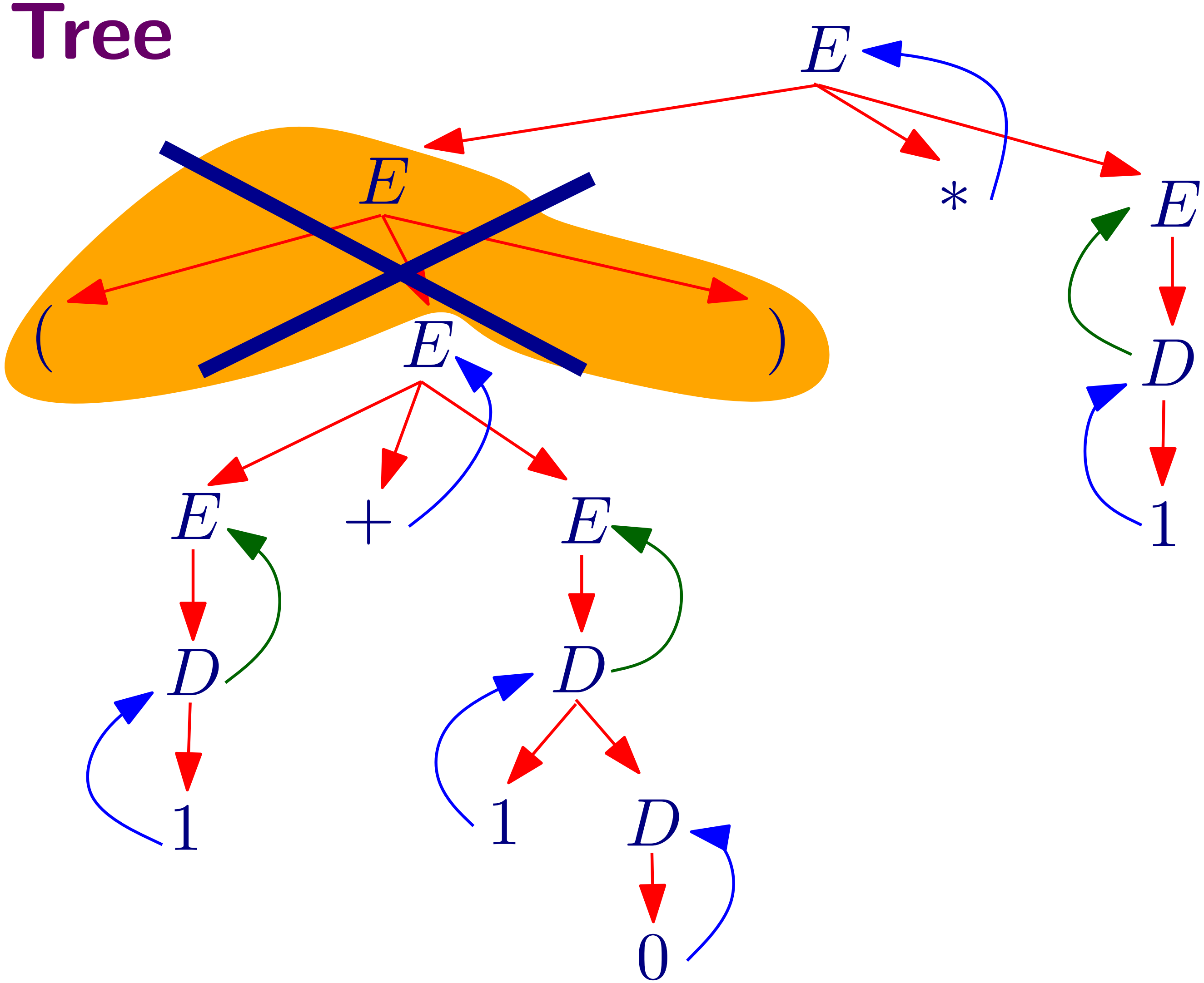
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Abstract Syntax Tree

Simplified version of the parse tree:

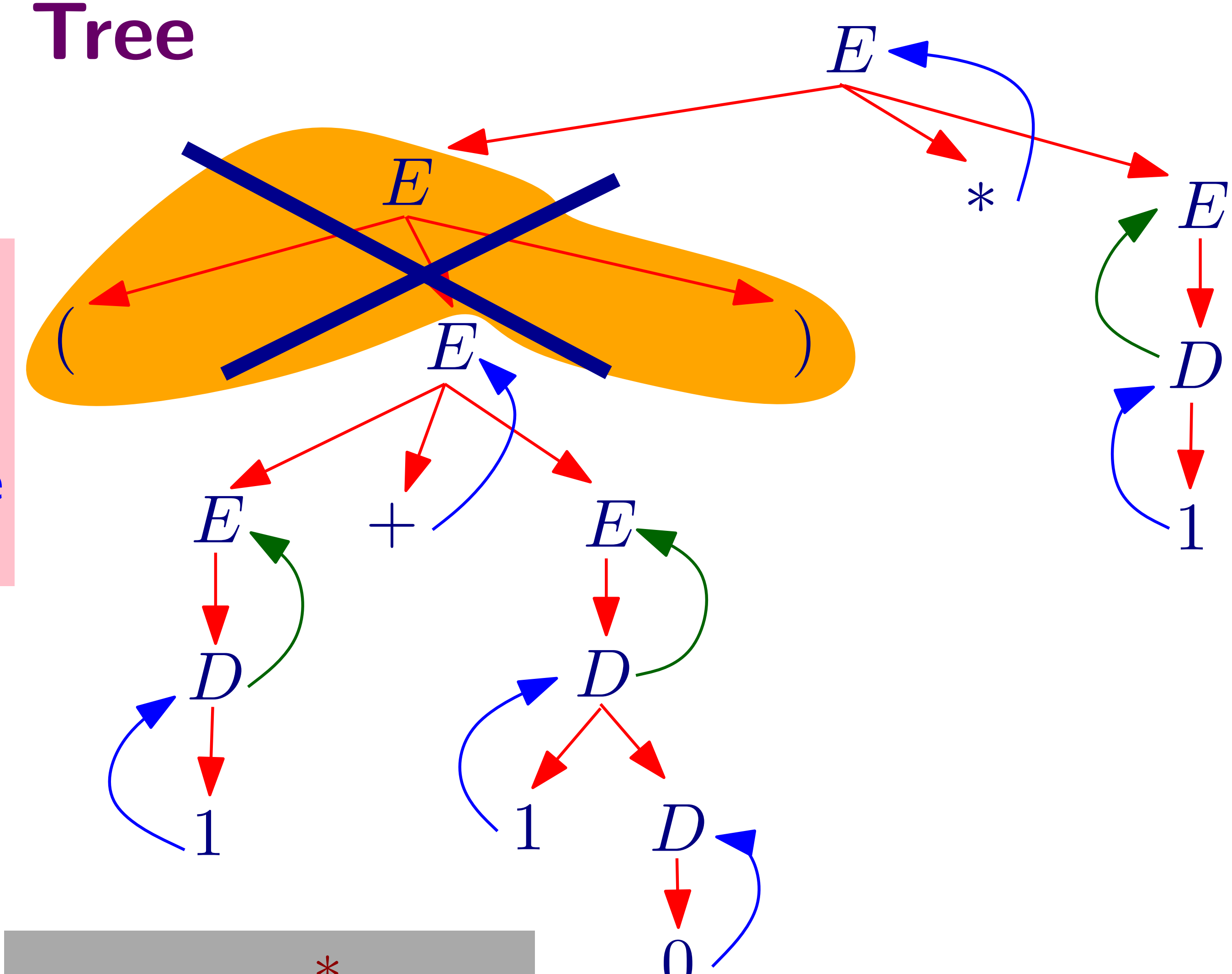
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



Abstract Syntax Tree

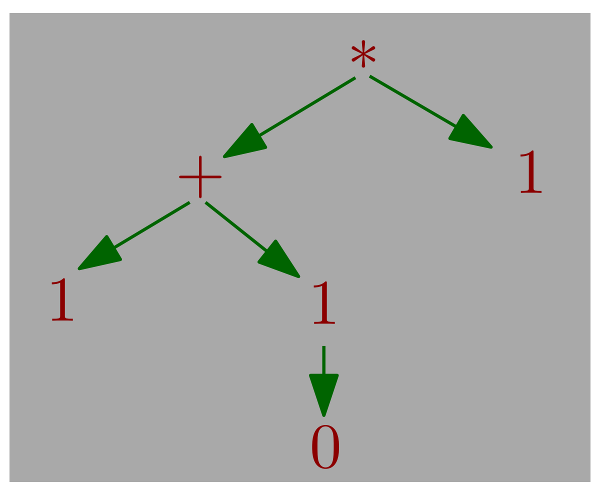
Simplified version of the parse tree:

- 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.



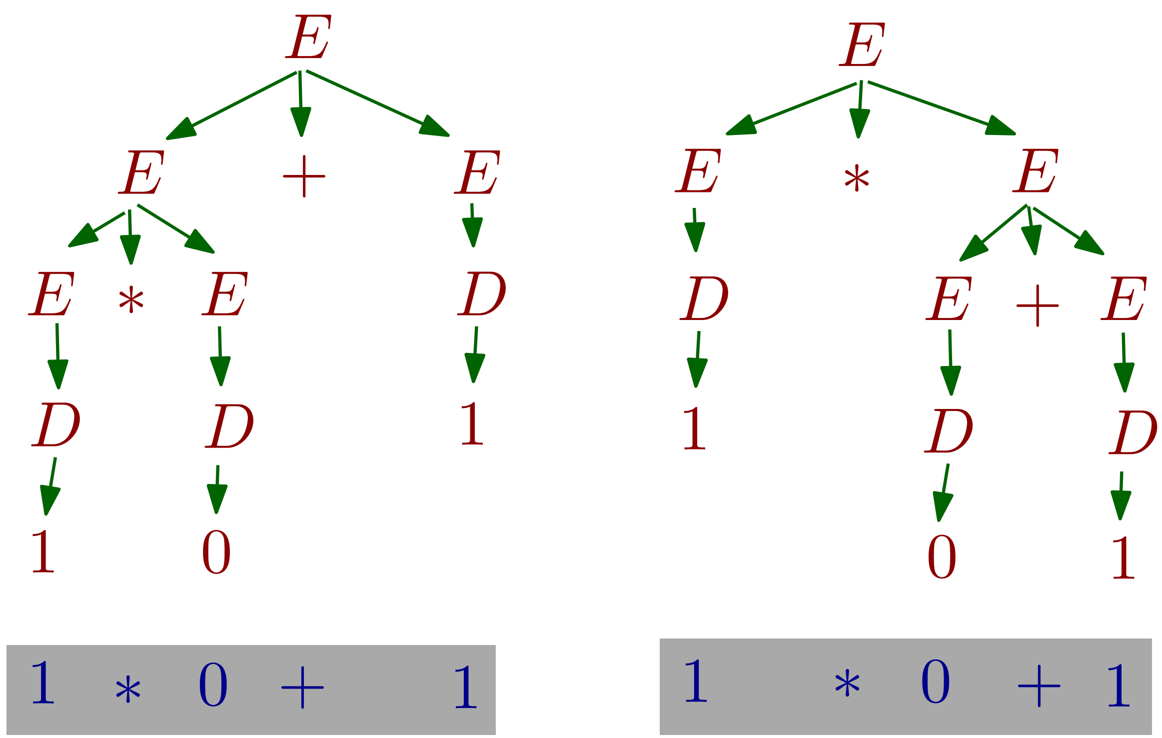
What Are Grammars Good For?

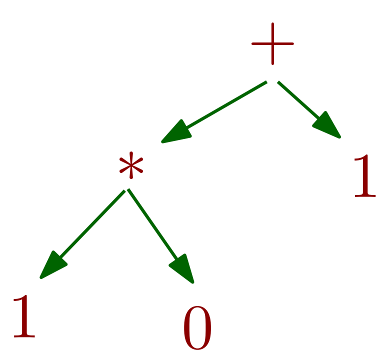
- Allow specification of (programming) languages – *generation*.
- Allow deciding whether a string belongs to the language or not – *acceptance*.
- Allow *syntactic analysis*.
- **Syntactic analysis:** building the Abstract Syntax Tree from a string or program.
- The AST can be used by compilers, program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

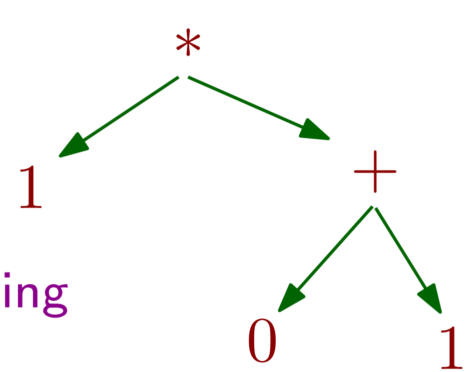
Ambiguity

Non-unique parse trees: *ambiguous grammar*

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





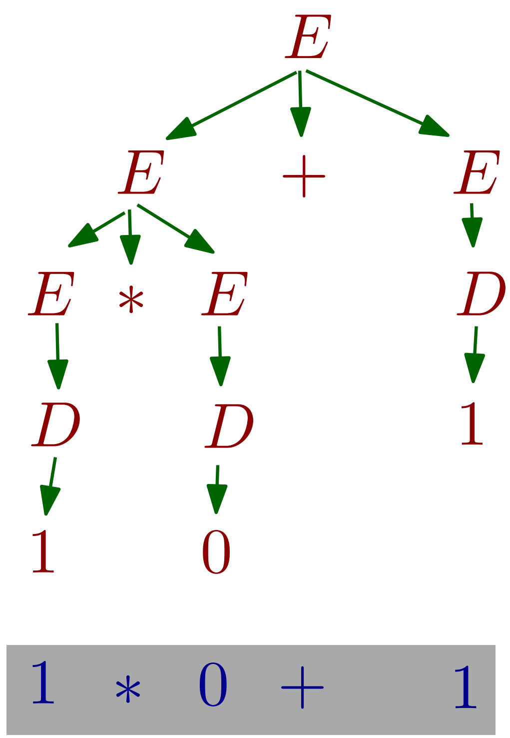


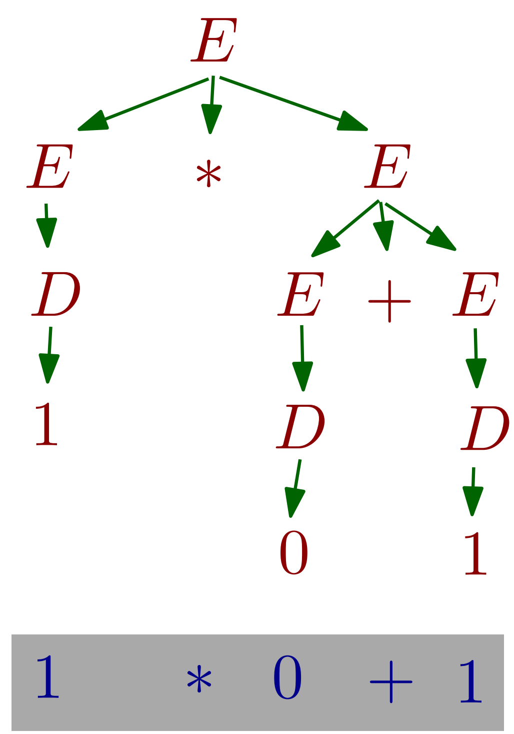
Corresponding ASTs

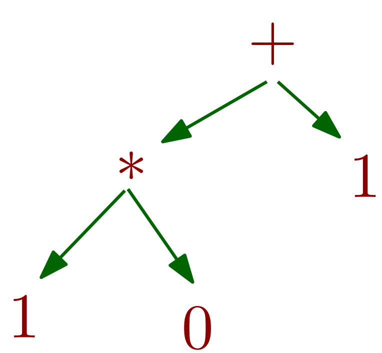
Ambiguity

Non-unique parse trees: *ambiguous grammar*

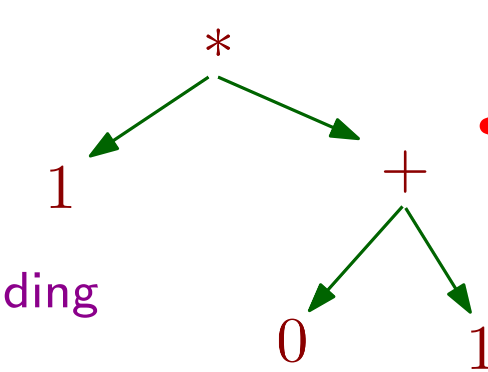
$$
\begin{array}{lcl}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0 \\
D & \rightarrow & 1
\end{array}
$$







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:

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

Backus-Naur Form

Original grammar:

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

BNF Meta-symbols

Equivalent BNF:

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

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:

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

Backus-Naur Form

Terminals are enclosed in simple or double quotes.

Original grammar:

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

Equivalent BNF:

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

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:

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

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:

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

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:

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

Either-or meta-symbol

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:

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

Factorization

Backus-Naur Form

Original grammar:

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

Equivalent BNF:

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

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:

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

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

Empty string: can appear inbetween any two terminals

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

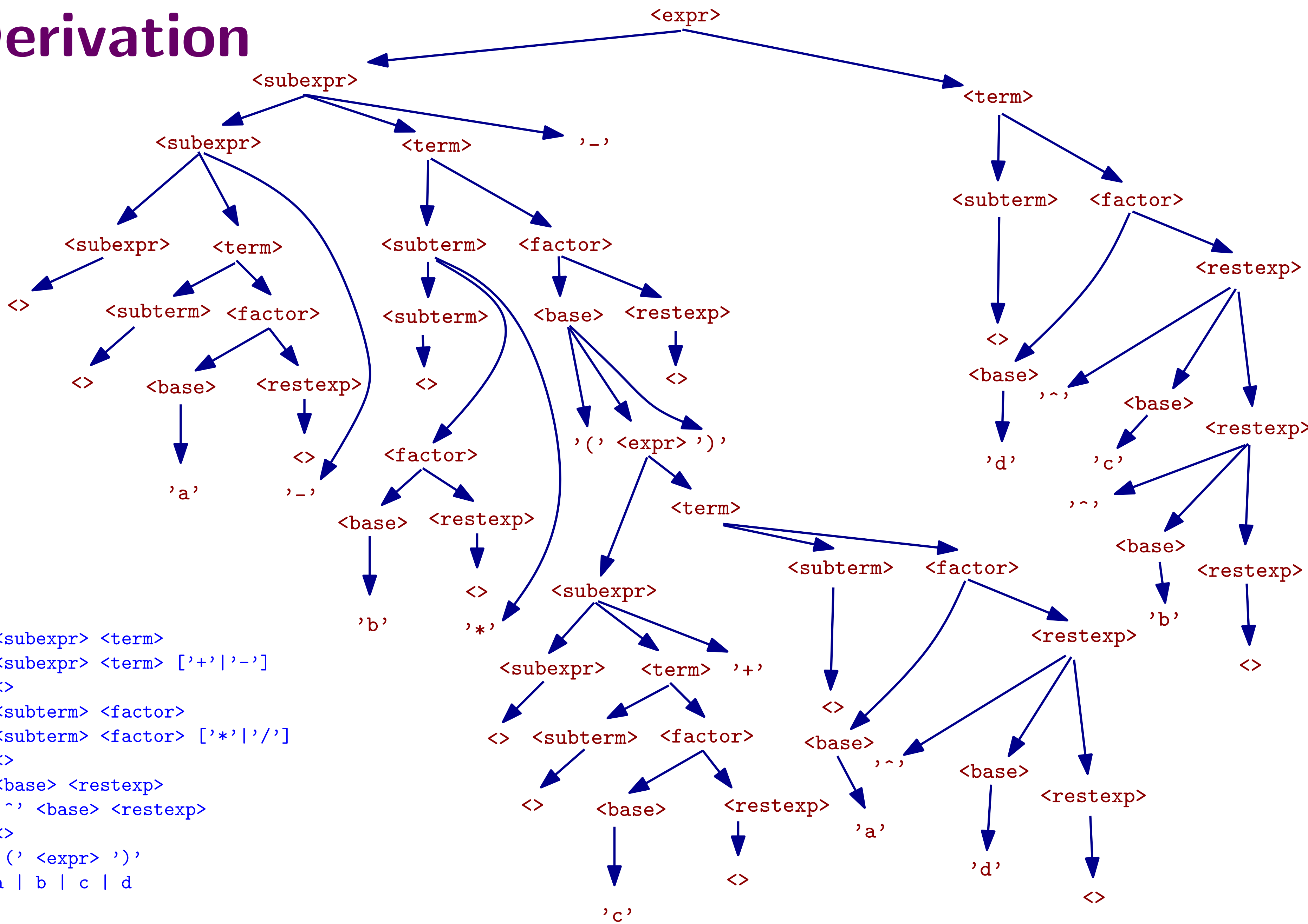
A Non-Ambiguous Grammar for Expressions

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
```

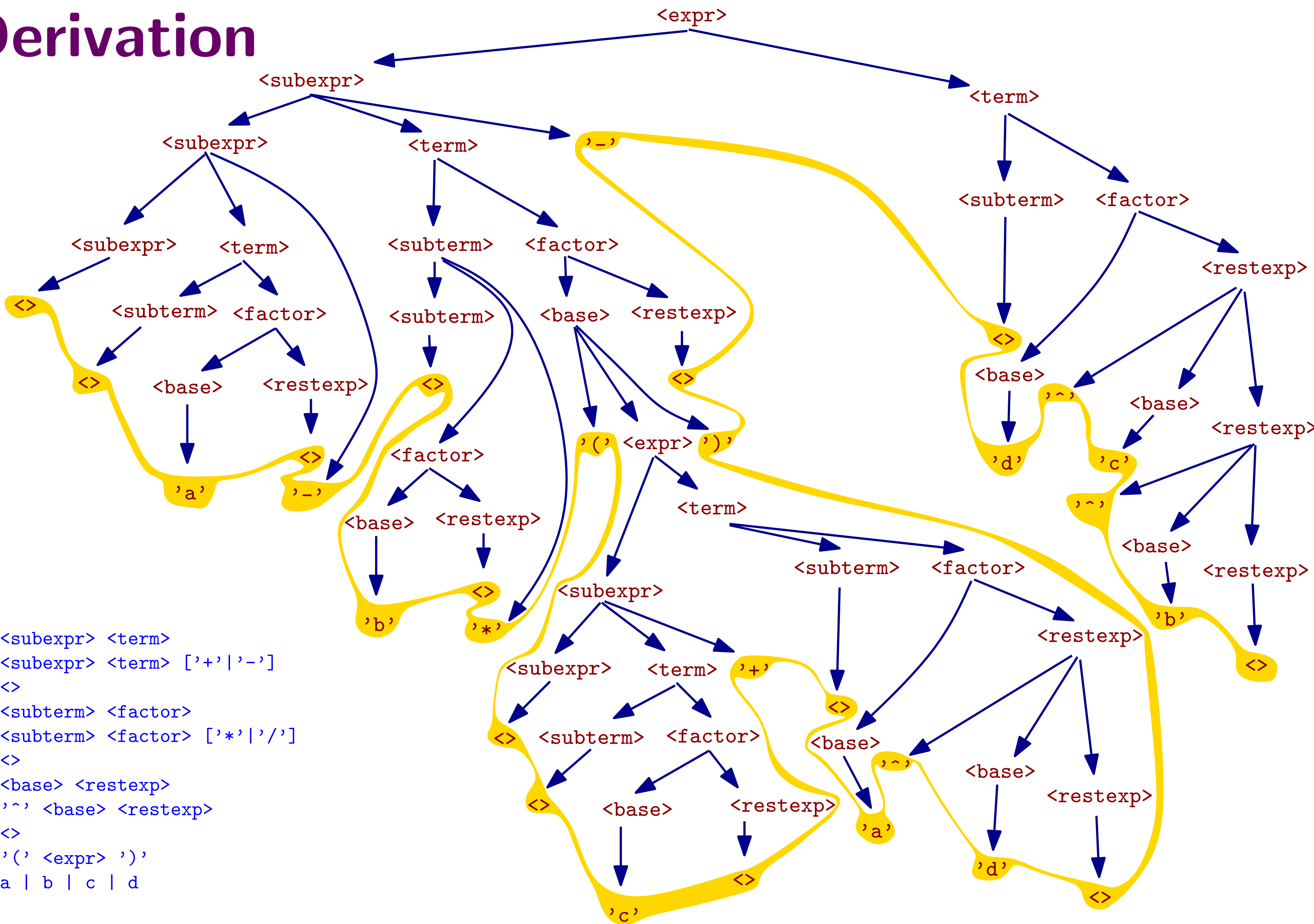
A Derivation



```
<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

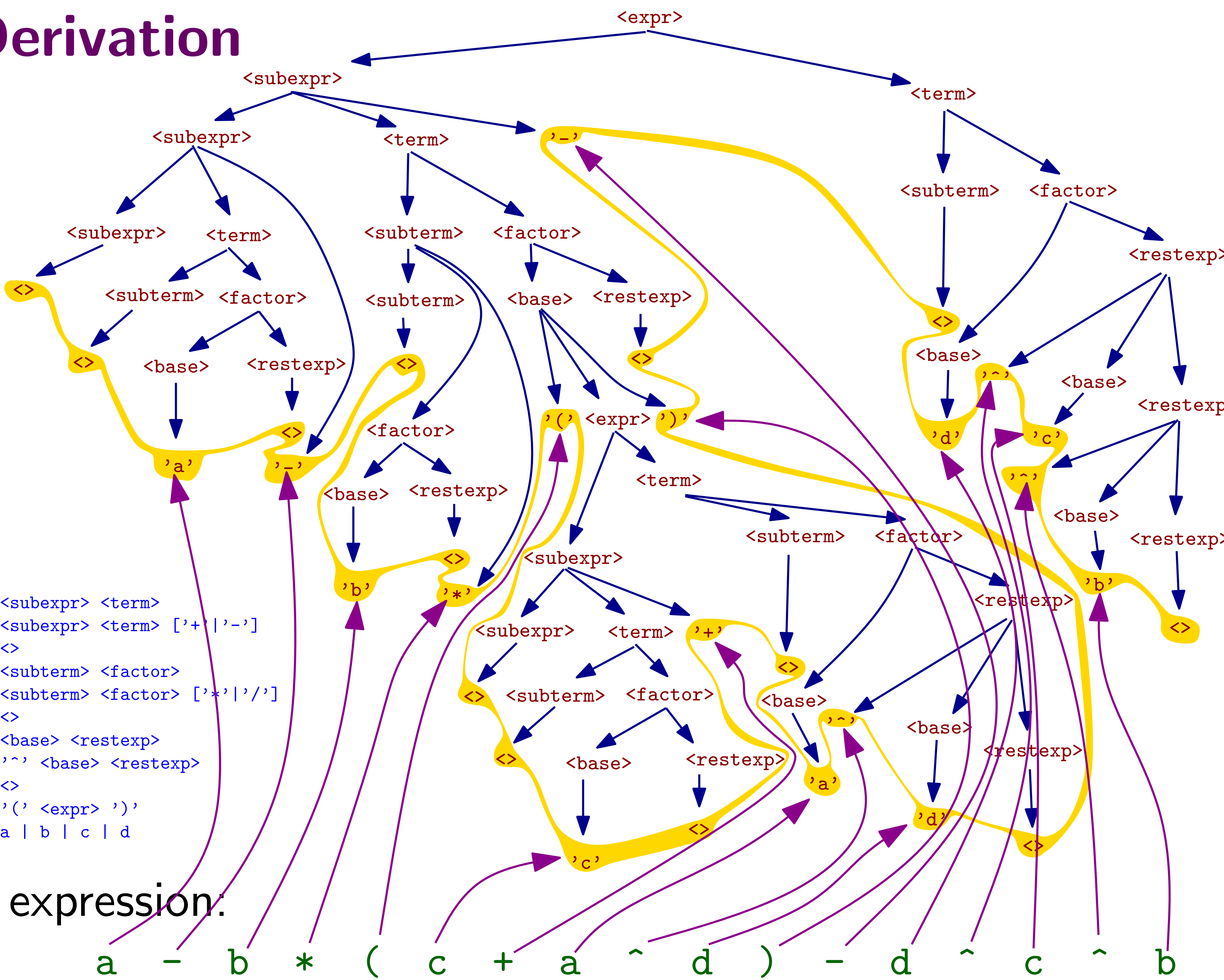



```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr>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

Parsed expression:

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

A Derivation

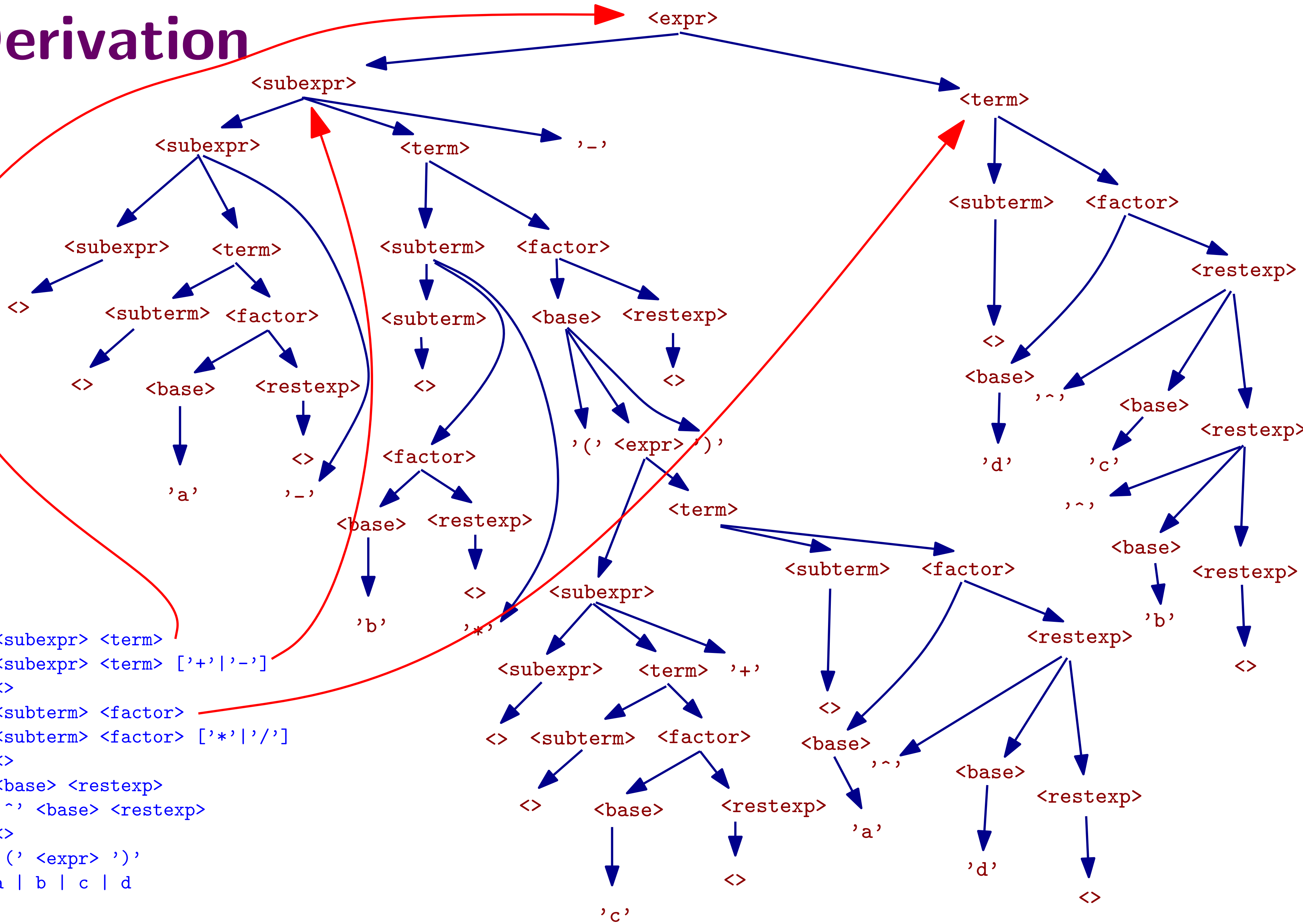



```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr>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

Parsed expression:

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

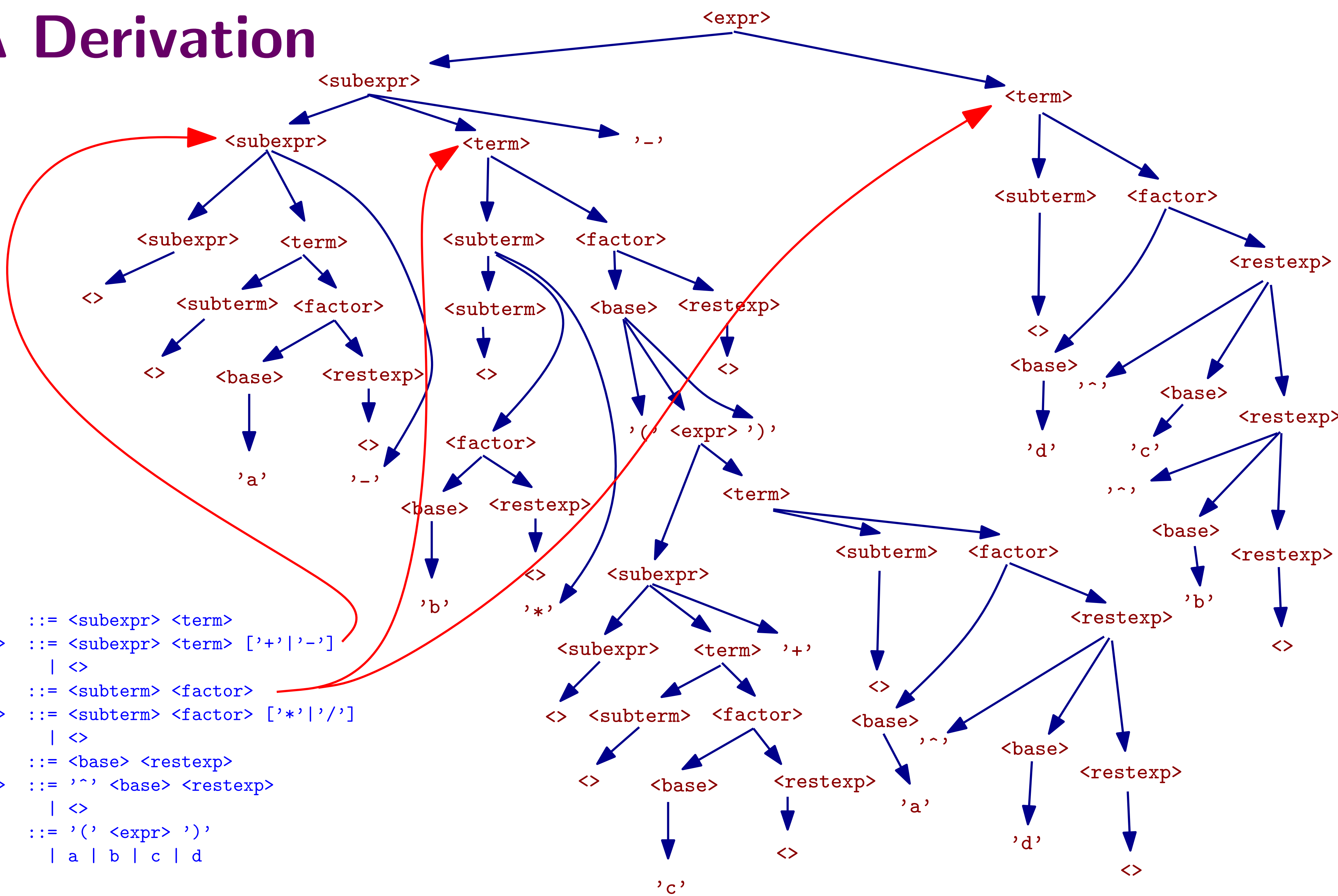
A Derivation




```
<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

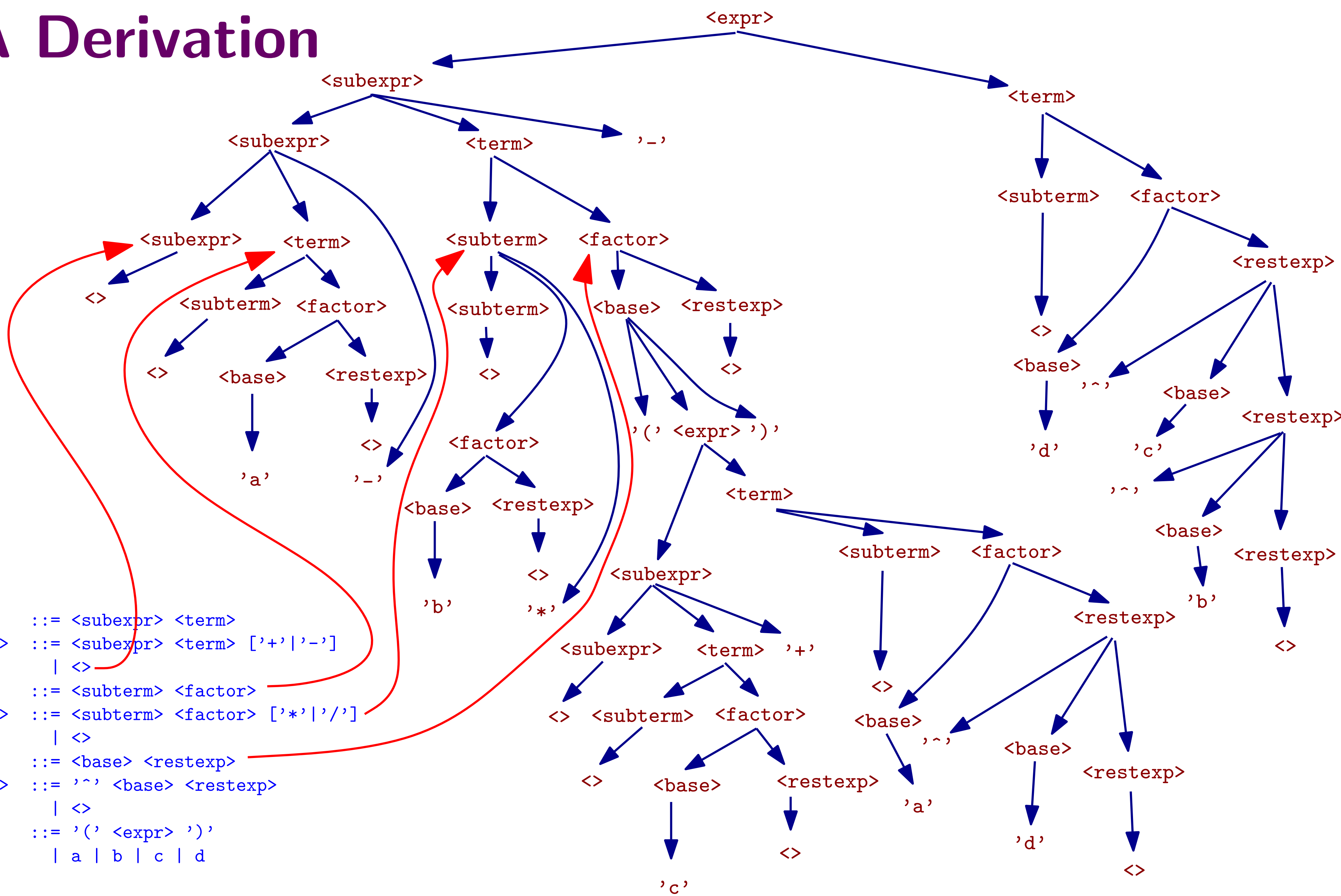
A Derivation




```
<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

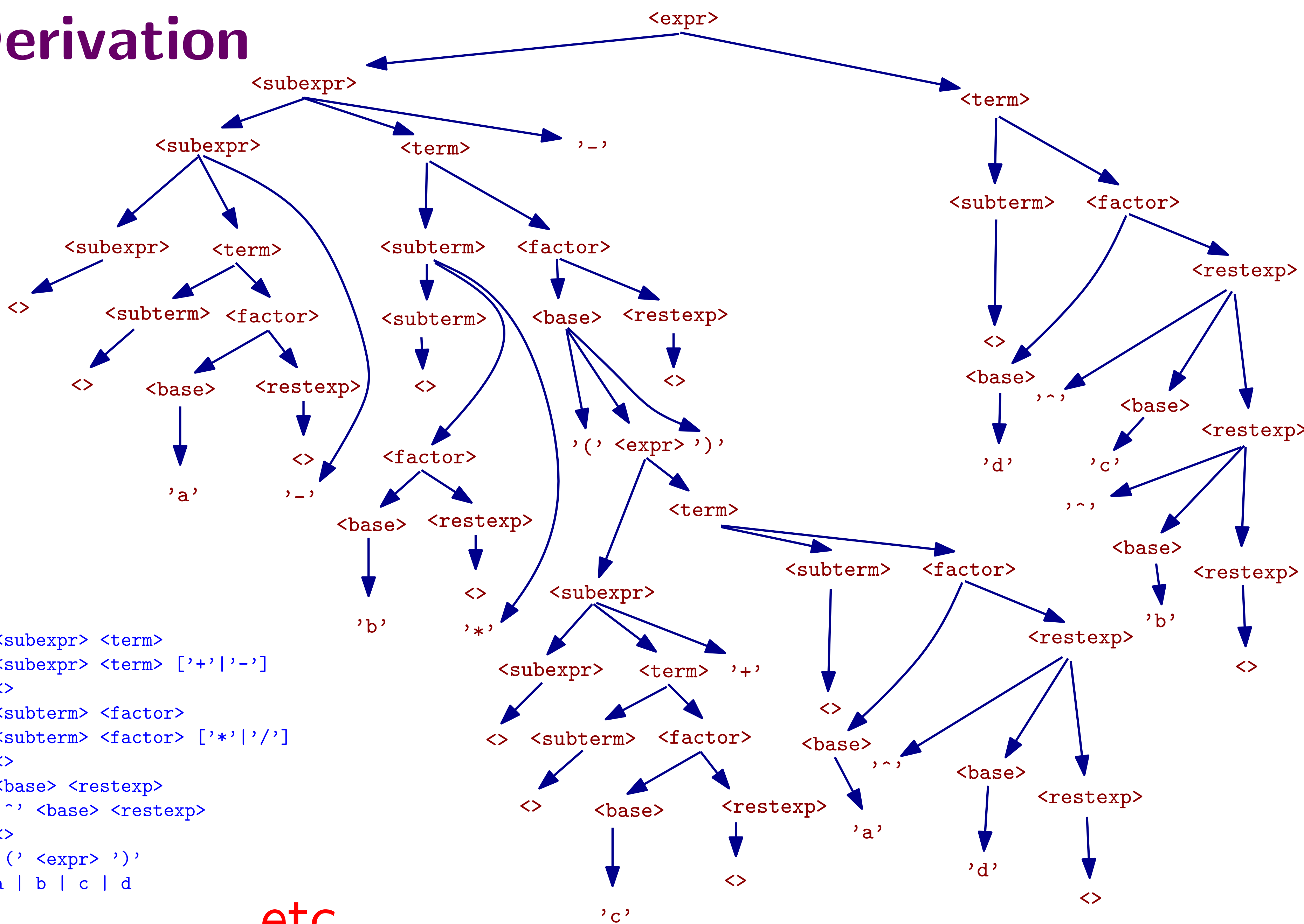
A Derivation




```
<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

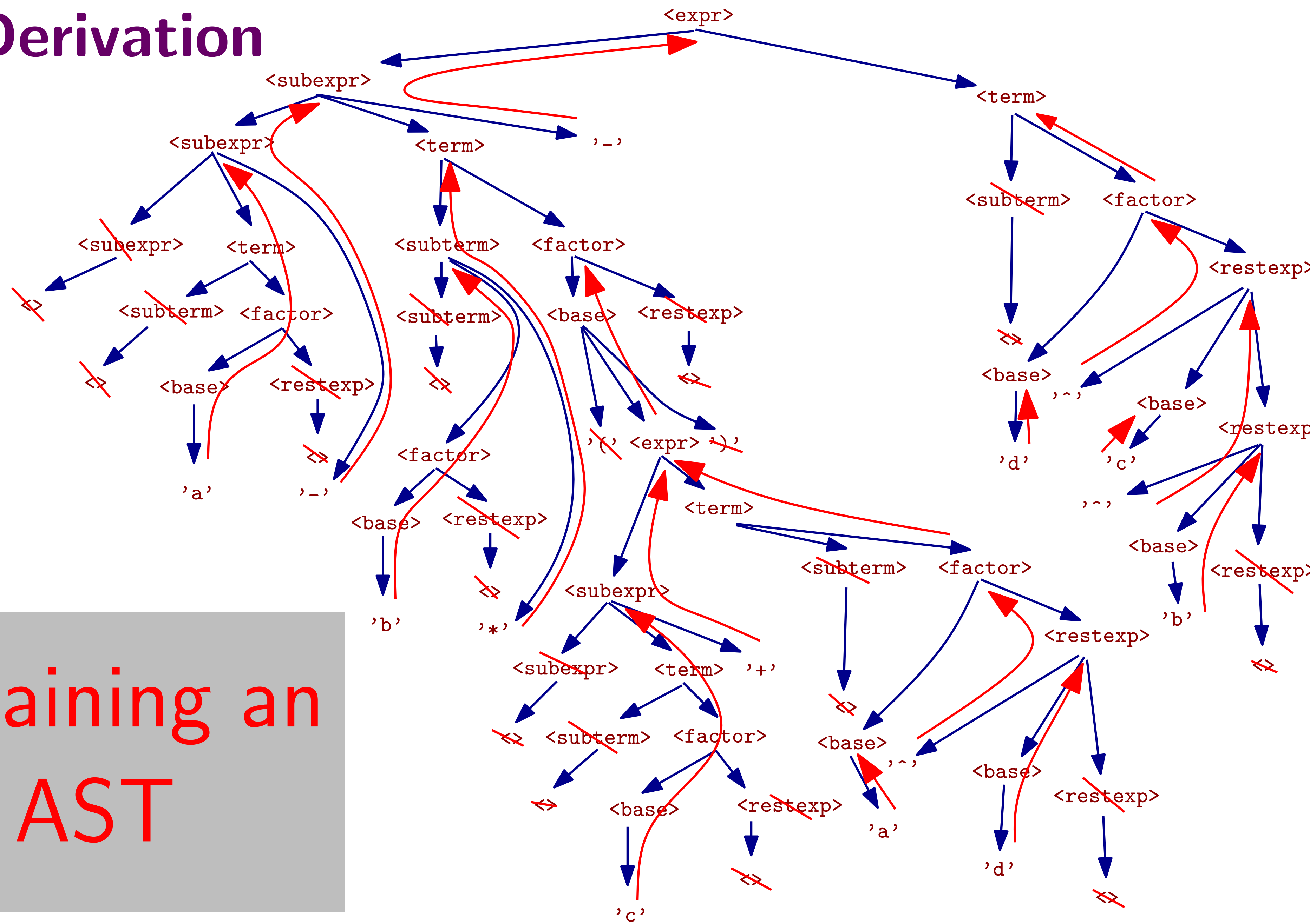



```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::=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

etc...

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

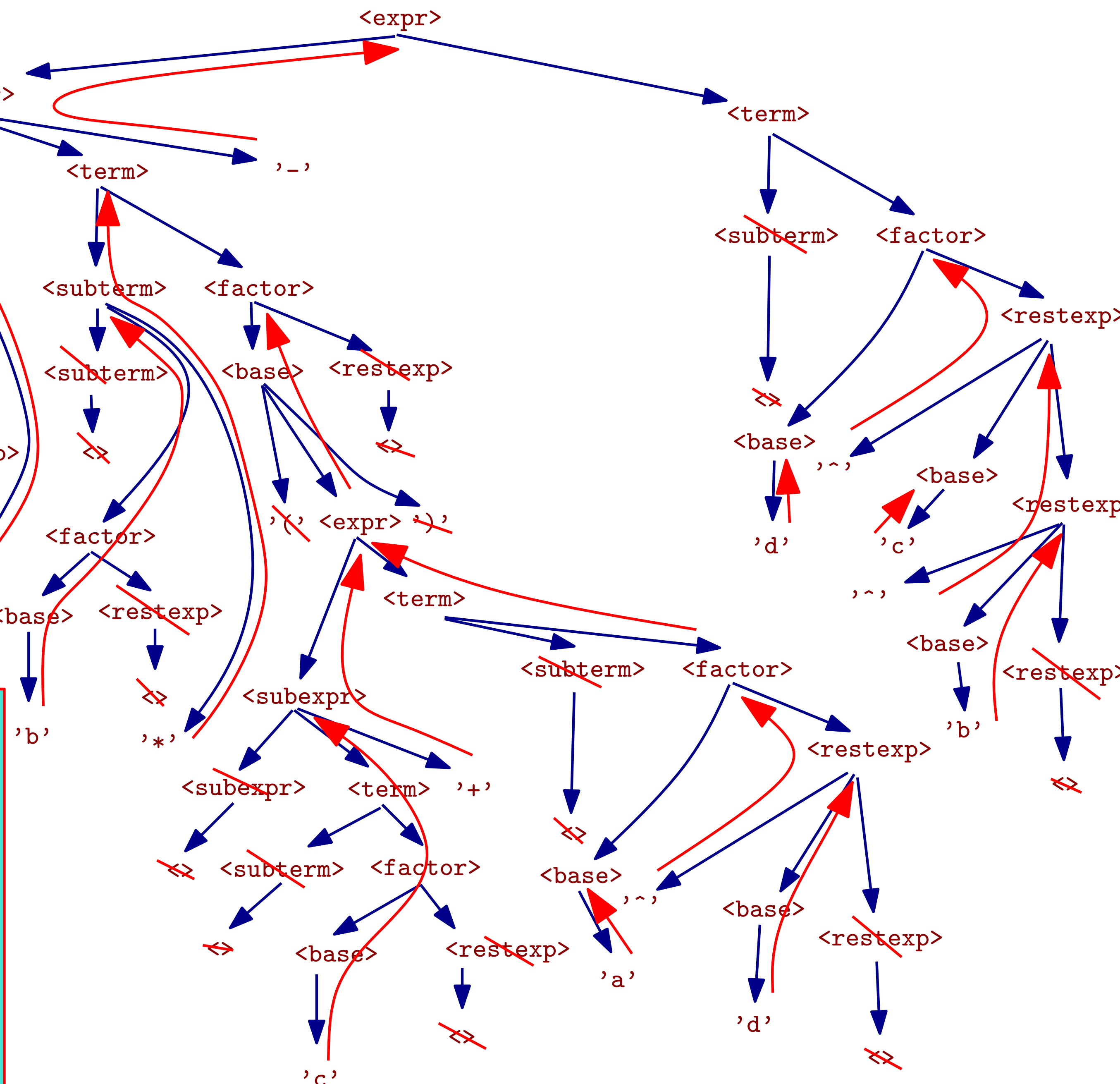
A Derivation



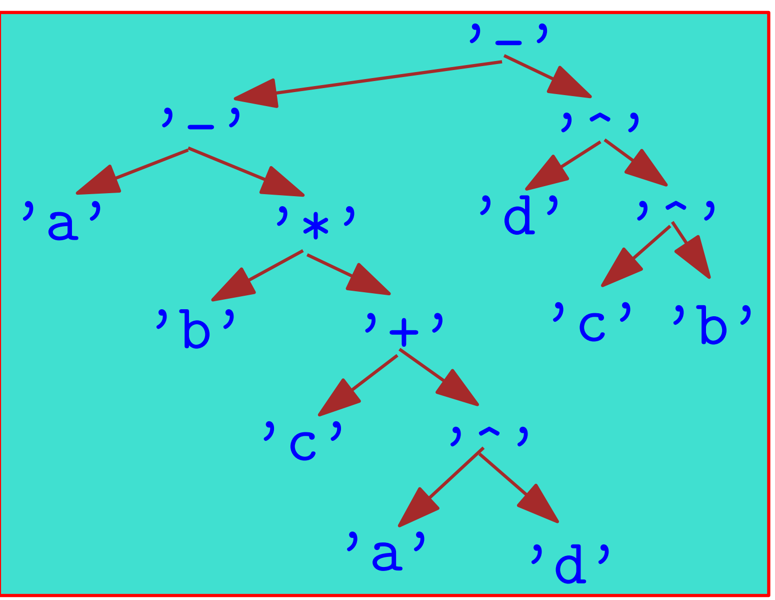
Obtaining an AST

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.

Stratification

- For syntactic analysis of programming languages, a 2-layered approach is used.
- The primitive elements of the language, such as *constants*, *identifiers*, *operators*, *keywords*, etc, *special purpose grammars are defined*.
- The start symbols of the special purpose grammars are then use as *terminals of a higher-level grammar*, which defines the language.
- Low-level parsing: *lexical analysis*
- High-level parsing: *syntactic analysis*
- Low-level parsing: the language has very little structure, grammars are in general very simple.
- High-level parsing: structure is important, and so is abstracting away from the lexical level; an identifier or a constant becomes a *terminal symbol of the higher-level grammar*.

Lexical Analysis Grammars

```
<PI> ::= <Digit> | <Digit> <PI>
<Digit> ::= '0' | '1' | ... | '9'

<String>  ::= '"' <Seq> '"'
<Seq>     ::= [ 'a' | 'b' | ... | 'z' |
                '0' | ... | '9' |
                '+' | '-' | ... ] <Seq>
            | <>

<Id>       ::= <Alph> <AlnumSeq>
<Alph>     ::= [ 'a' | ... | 'z' | 'A' | ... | 'Z' | '_' ]
<Alnum>    ::= <Alph> | <Digit>
<AlnumSeq> ::= <Alnum> <AlnumSeq>
             | <>
```

Lexical Analysis Grammars

Right recursive!

```
<PI> ::= <Digit> | <Digit> <PI>
<Digit> ::= '0' | '1' | ... | '9'

<String>  ::= '"' <Seq> '"'
<Seq>     ::= [ 'a' | 'b' | ... | 'z' |
                '0' | ... | '9' |
                '+' | '-' | ... ] <Seq>
            | <>

<Id>        ::= <Alph> <AlnumSeq>
<Alph>      ::= [ 'a' | ... | 'z' | 'A' | ... | 'Z' | '_' ]
<Alnum>     ::= <Alph> | <Digit>
<AlnumSeq> ::= <Alnum> <AlnumSeq>
             | <>
```

Grammars whose recursion is always in the rightmost position in the body of the rule is called *regular*.

Lexical Analysis Grammars

Right recursive!

```
<PI> ::= <Digit> | <Digit> <PI>
<Digit> ::= '0' | '1' | ... | '9'

<String>  ::= '"' <Seq> '"'
<Seq>     ::= [ 'a' | 'b' | ... | 'z' |
                '0' | ... | '9' |
                '+' | '-' | ... ] <Seq>
            | <>

<Id>        ::= <Alph> <AlnumSeq>
<Alph>      ::= [ 'a' | ... | 'z' | 'A' | ... | 'Z' | '_' ]
<Alnum>     ::= <Alph> | <Digit>
<AlnumSeq>  ::= <Alnum> <AlnumSeq>
              | <>
```

Oversimplification, doesn't consider mutual recursion!

Grammars whose recursion is always in the rightmost position in the body of the rule is called *regular*.

Regular Languages

A *right regular grammar* (also called *right linear grammar*) is a grammar (N, Σ, P, S) such that all the production rules in P are one of the following forms:

(a) $B \rightarrow a$, where $B \in N$ and $a \in \Sigma$.

(b) $B \rightarrow aC$, where $B, C \in N$ and $a \in \Sigma$.

(c) $B \rightarrow \epsilon$, where $B \in N$ and ϵ is the empty string.

Regular Languages

A *right regular grammar* (also called *right linear grammar*) is a grammar (N, Σ, P, S) such that all the production rules in P are one of the following forms:

(a) $B \rightarrow a$, where $B \in N$ and $a \in \Sigma$.
(b) $B \rightarrow aC$, where $B, C \in N$ and $a \in \Sigma$.
(c) $B \rightarrow \epsilon$, where $B \in N$ and ϵ is the empty string.

A similar definition exists for *left regular grammars*. Given a left regular grammar G_L, there exists a right regular grammar G_R such that $\mathcal{L}(G_L) = \mathcal{L}(G_R)$.

Regular Languages

A *right regular grammar* (also called *right linear grammar*) is a grammar (N, Σ, P, S) such that all the production rules in P are one of the following forms:

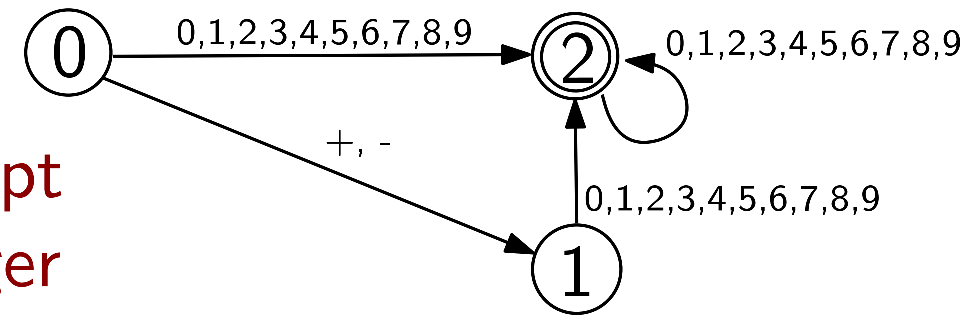
(a) $B \to a$, where $B \in N$ and $a \in \Sigma$.
(b) $B \to aC$, where $B, C \in N$ and $a \in \Sigma$.
(c) $B \to \epsilon$, where $B \in N$ and ϵ is the empty string.

A similar definition exists for *left regular grammars*. Given a left regular grammar G_L, there exists a right regular grammar G_R such that $\mathcal{L}(G_L) = \mathcal{L}(G_R)$.

We shall only focus on *right regular grammars*. All languages generated by right or left regular grammars are called *regular languages*.

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

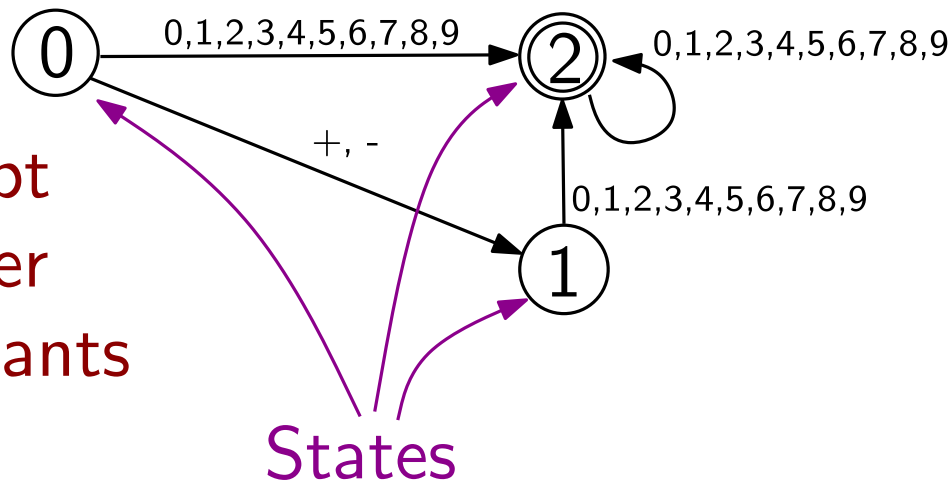


Accept integer constants

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

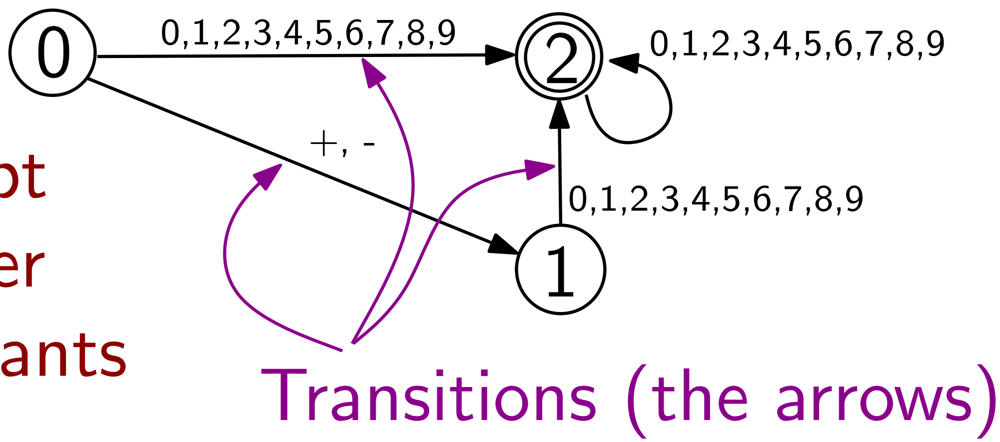
Accept
integer
constants



Deterministic Finite Automata

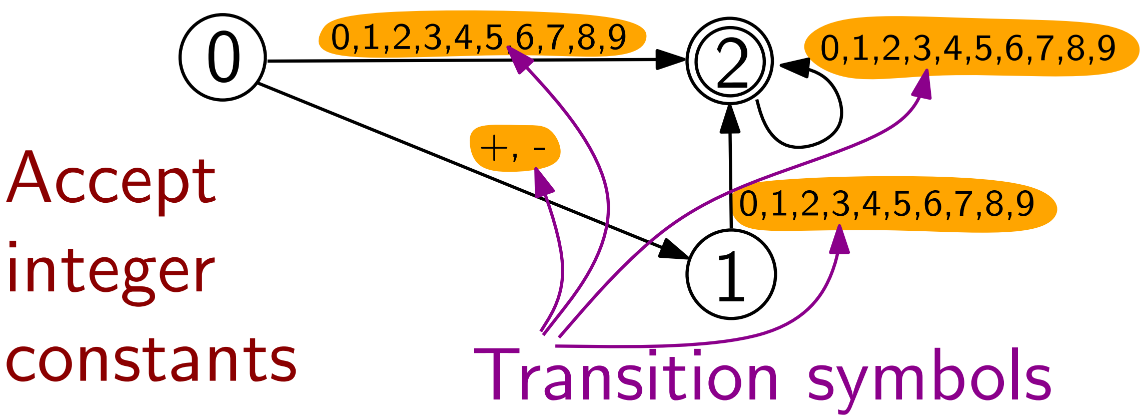
Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

Accept integer constants



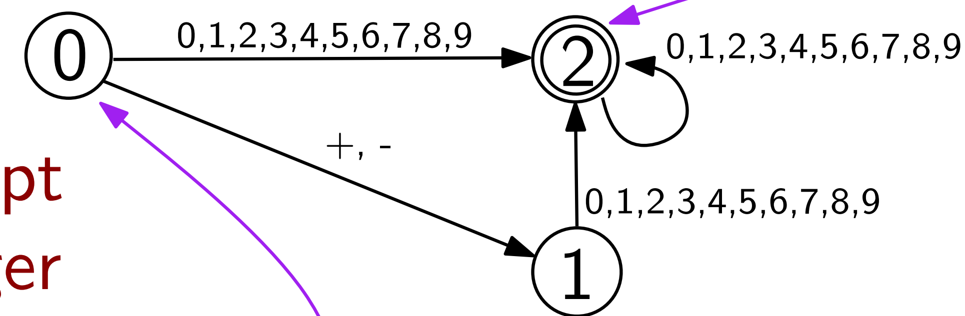
Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Final state

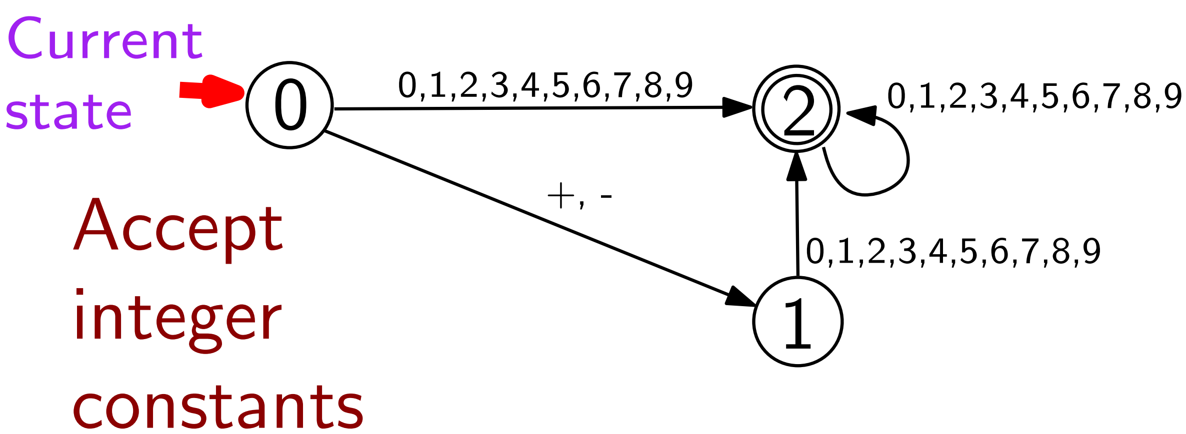
Accept integer constants

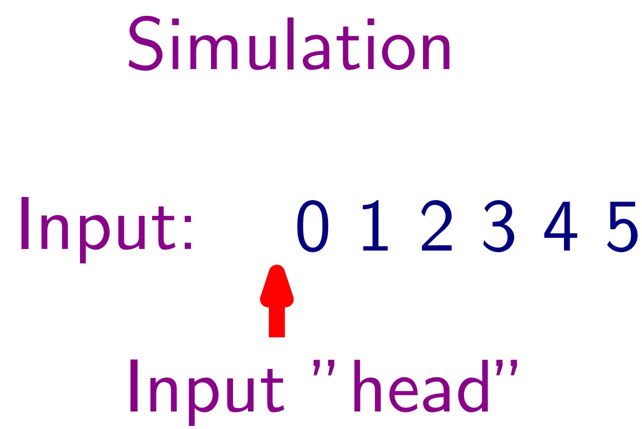
Initial state

(usually labelled 0 or q_0, and appearing in the top left corner of diagram)

Deterministic Finite Automata

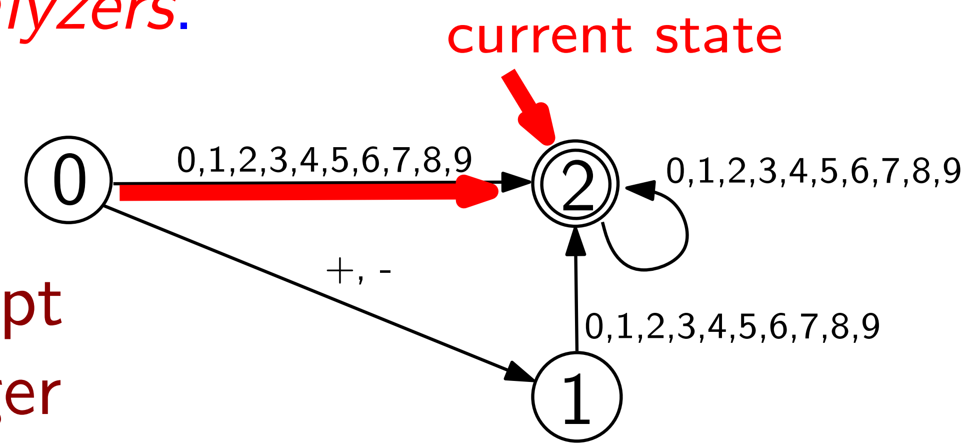
Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.





Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Accept integer constants

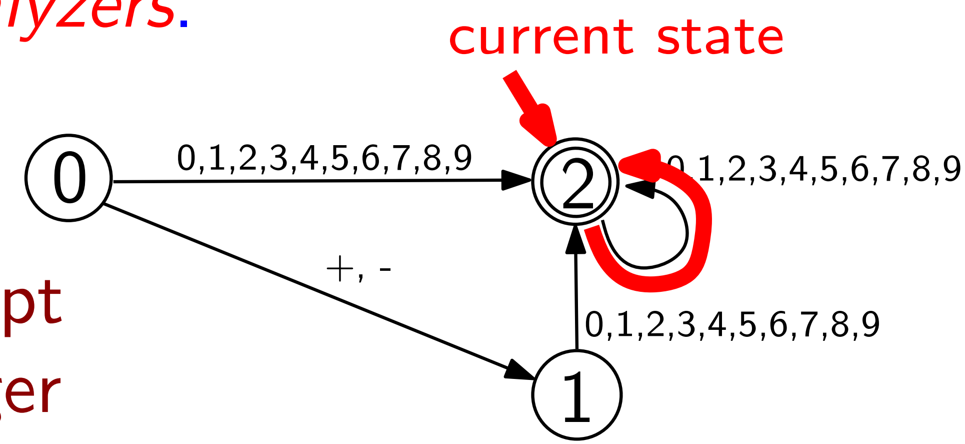
Simulation

Input: 0 1 2 3 4 5

Input "head"

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Accept
integer
constants

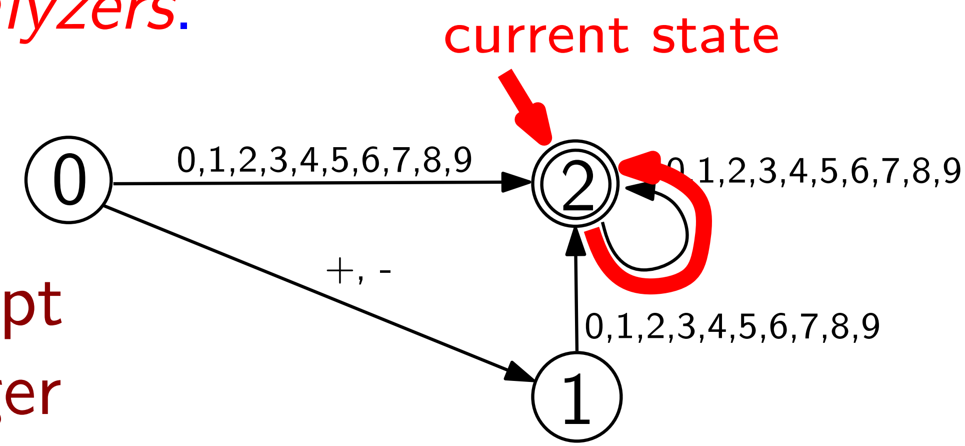
Simulation

Input: 0 1 2 3 4 5

Input "head"

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Accept integer constants

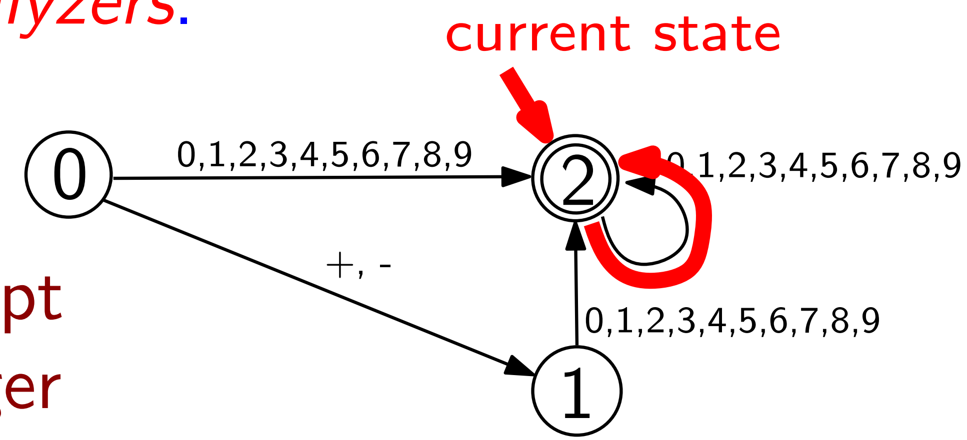
Simulation

Input: 0 1 2 3 4 5

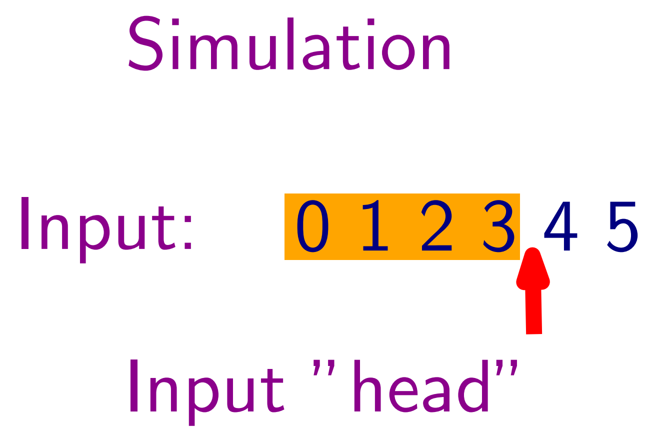
Input "head"

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

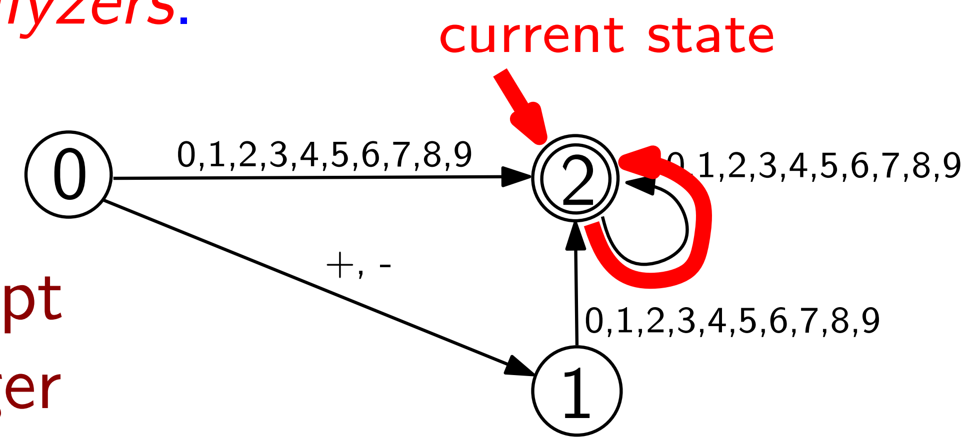


Accept integer constants



Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Accept integer constants

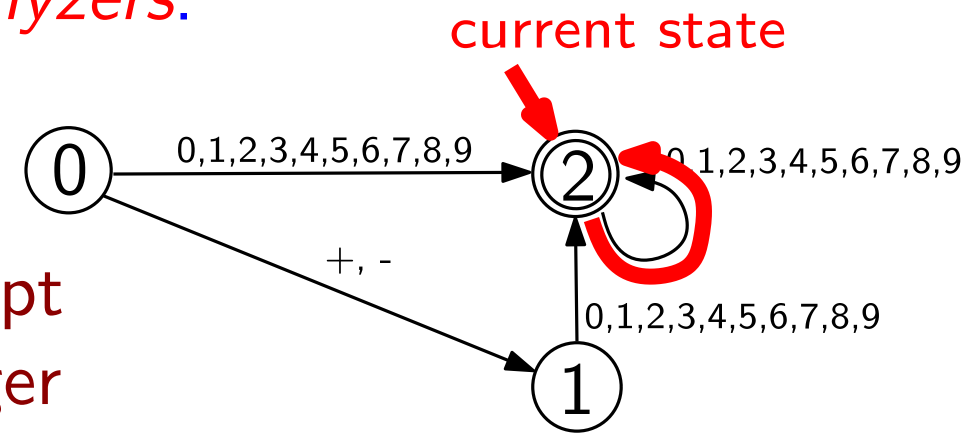
Simulation

Input: 0 1 2 3 4 5

Input "head"

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Accept integer constants

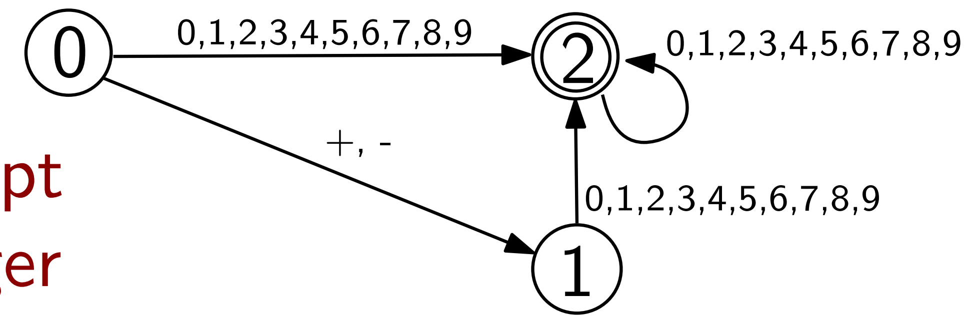
Simulation

Input "head"

If at the end of the input, or whenever the "head" encounters a symbol that is not a current transition symbol, the machine is in a final state, the string is *"accepted"*. Otherwise, the string is *"rejected"*.

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



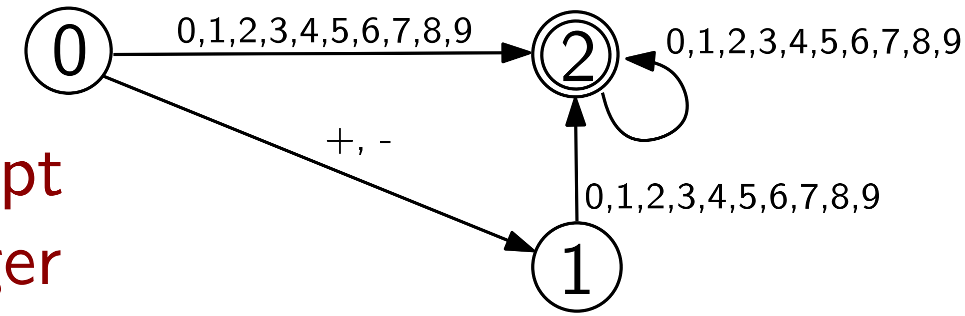
Accept integer constants

Equivalent regular grammar:

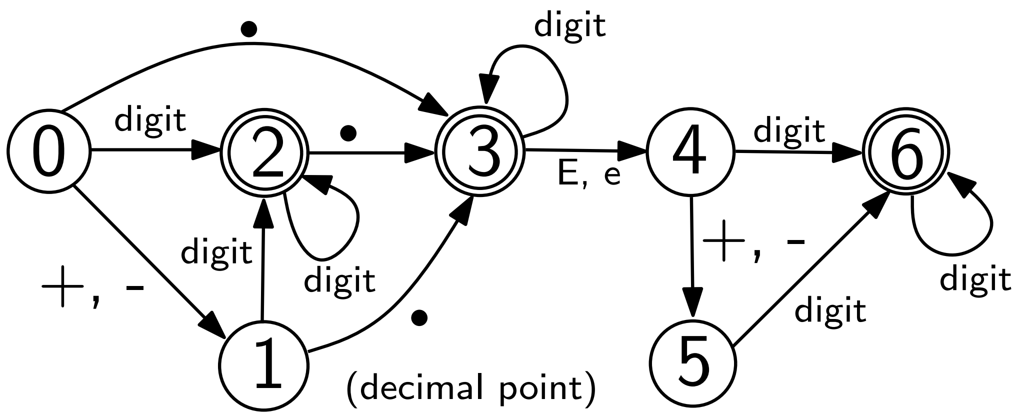
```
<0> ::= [ '0' | ... | '9' ] <2>
      | [ '+' | '-' ] <1>
<1> ::= [ '0' | ... | '9' ] <2>
<2> ::= [ '0' | ... | '9' ] <2>
      | <>
```

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.



Accept integer constants

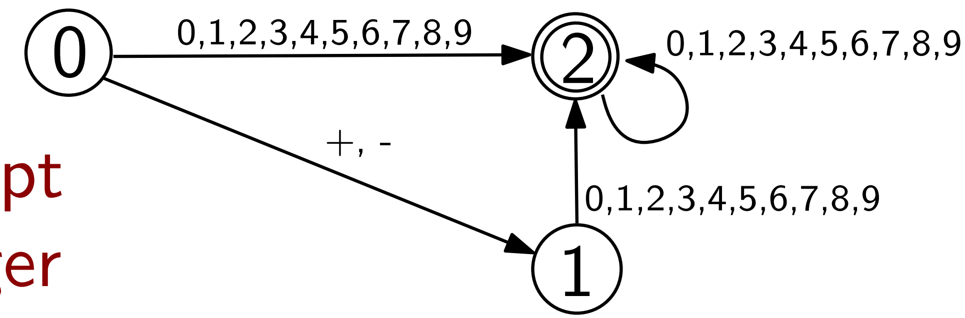


Accept real constants

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

Accept integer constants

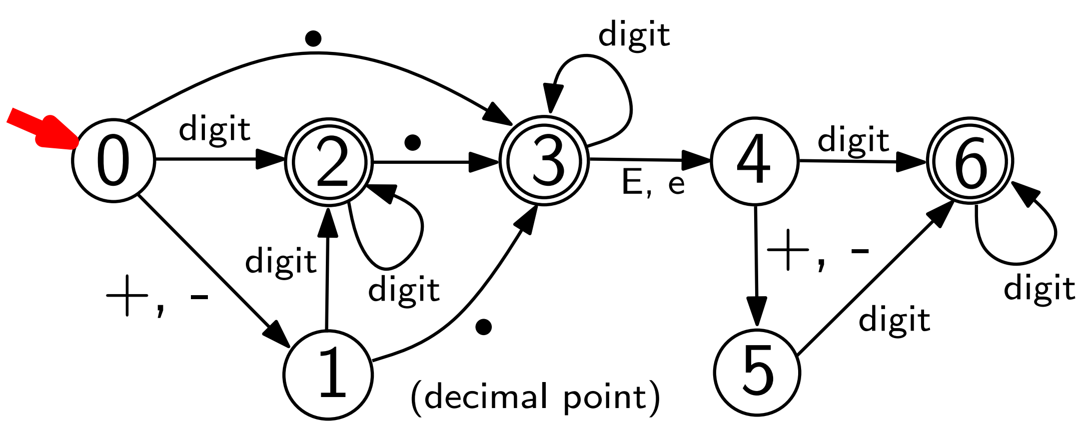


Simulation

Input: - 1 . 2 E + 3

Input "head"

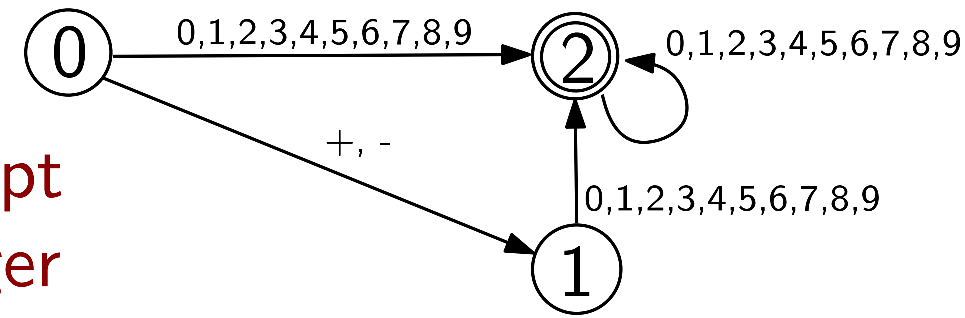
Accept real constants



Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

Accept integer constants

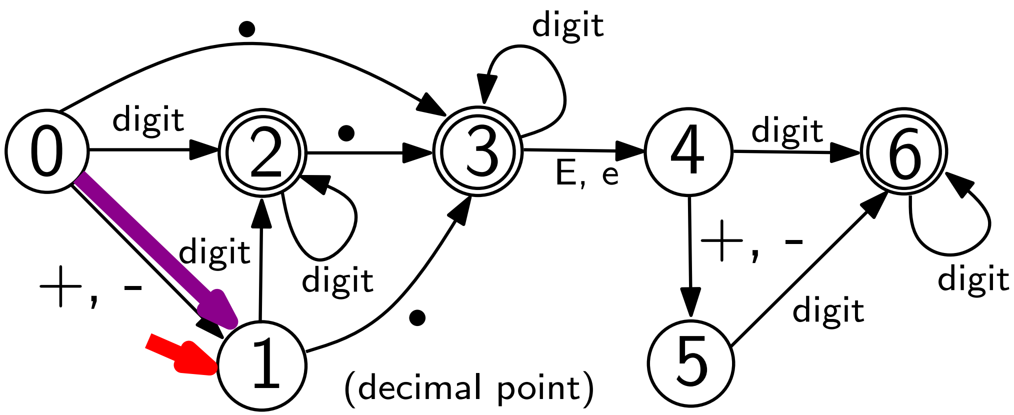


Simulation

Input: - 1 . 2 E + 3

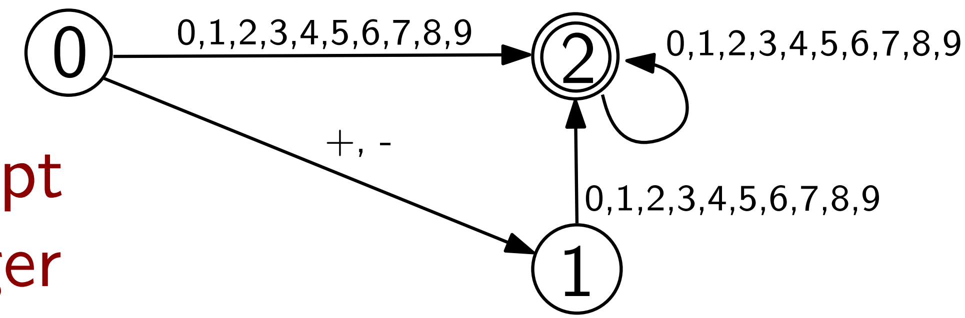
Input "head"

Accept real constants



Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

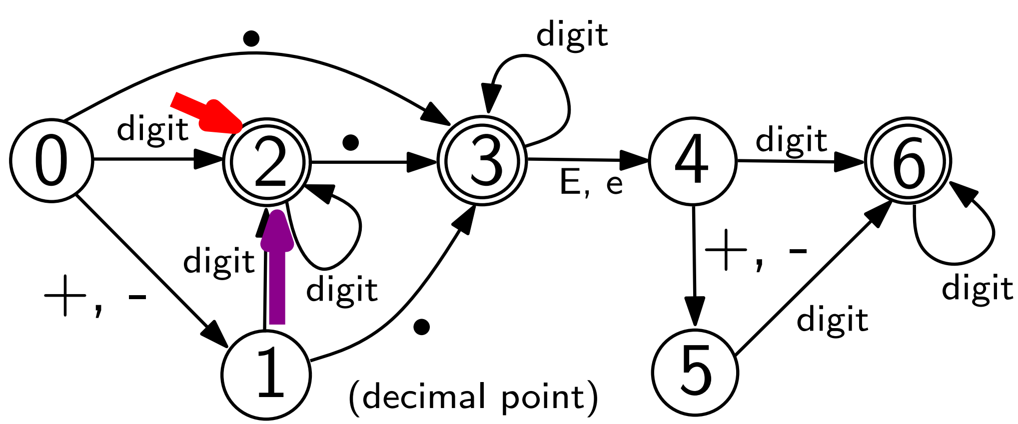


Accept integer constants

Simulation

Input: - 1 . 2 E + 3

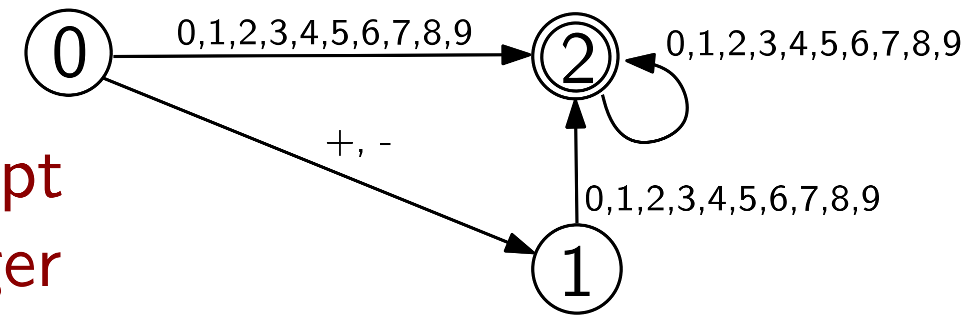
Input "head"



Accept real constants

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

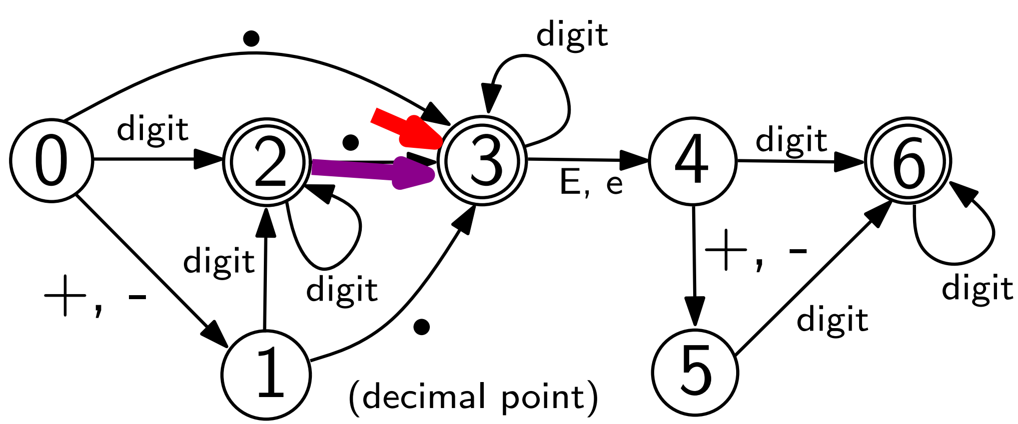


Accept integer constants

Simulation

Input: - 1 . 2 E + 3

Input "head"

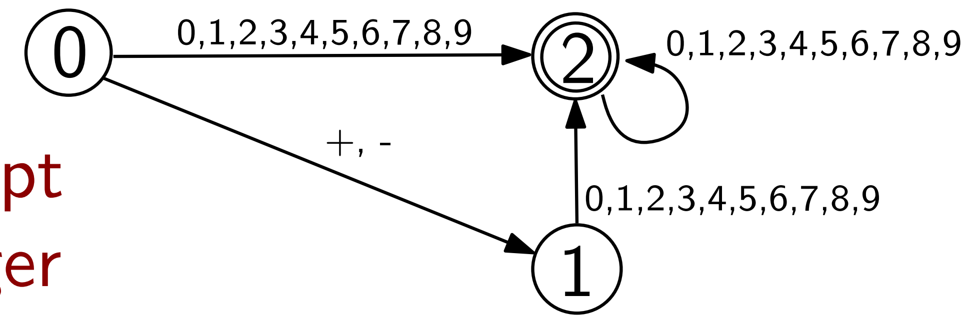


Accept real constants

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

Accept integer constants

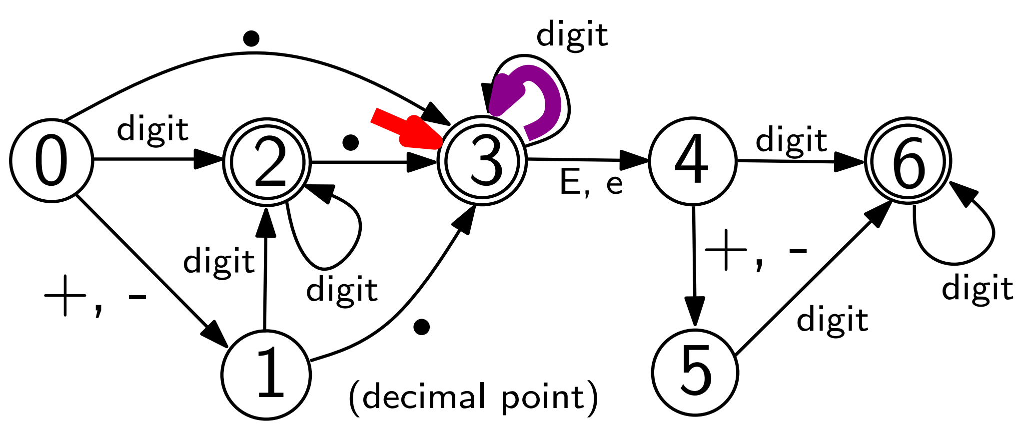


Simulation

Input: - 1 . 2 E + 3

Input "head"

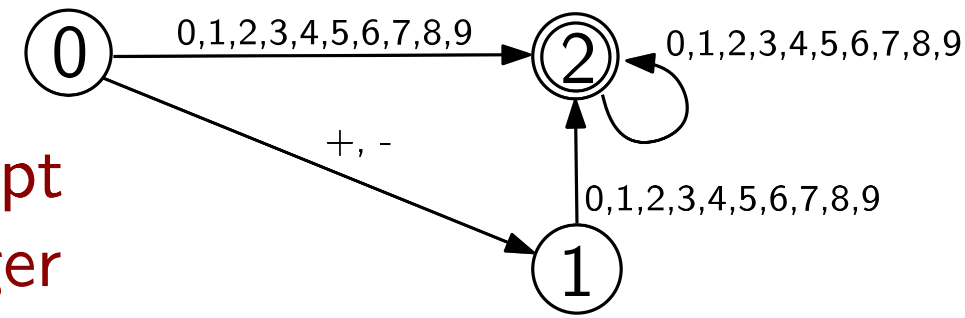
Accept real constants



Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

Accept integer constants

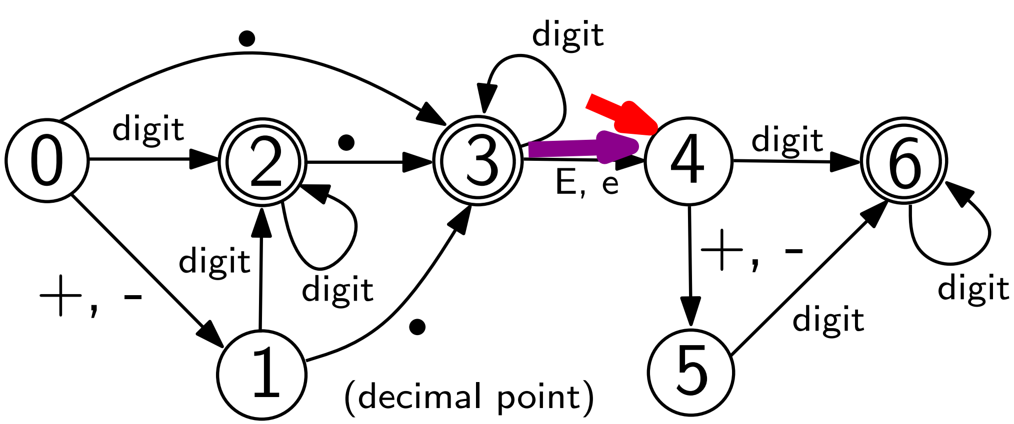


Simulation

Input: - 1 . 2 E + 3

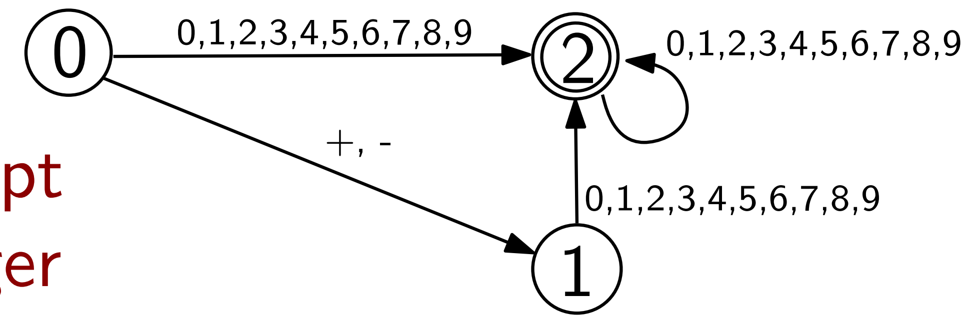
Input "head"

Accept real constants



Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

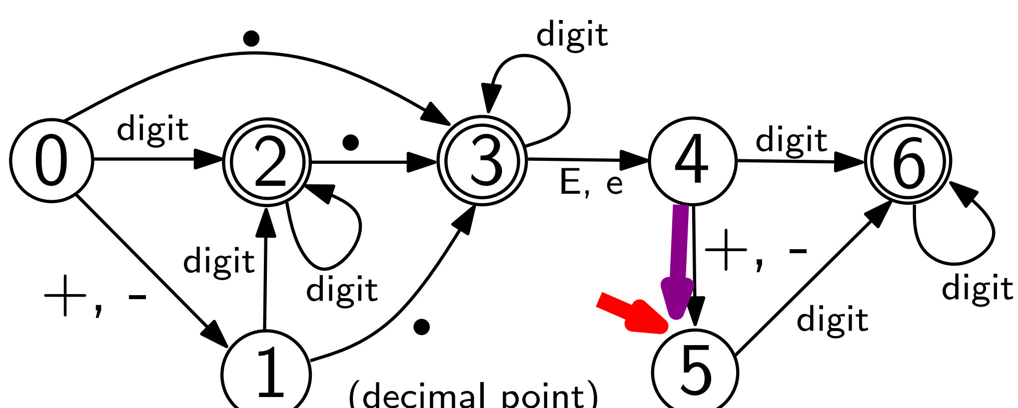


Accept integer constants

Simulation

Input: - 1 . 2 E + 3

Input "head"

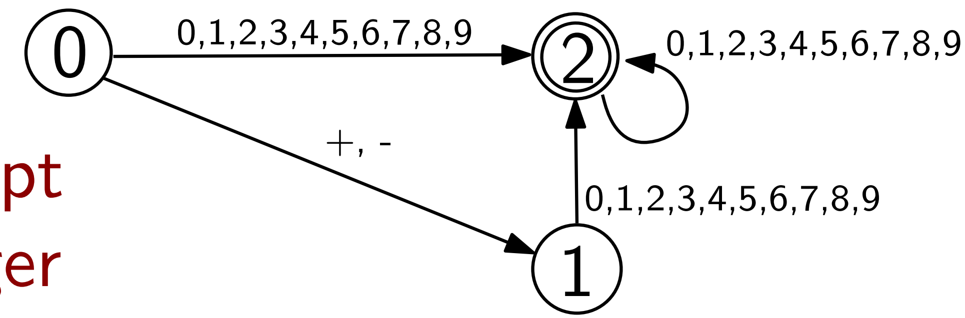


Accept real constants

Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

Accept integer constants

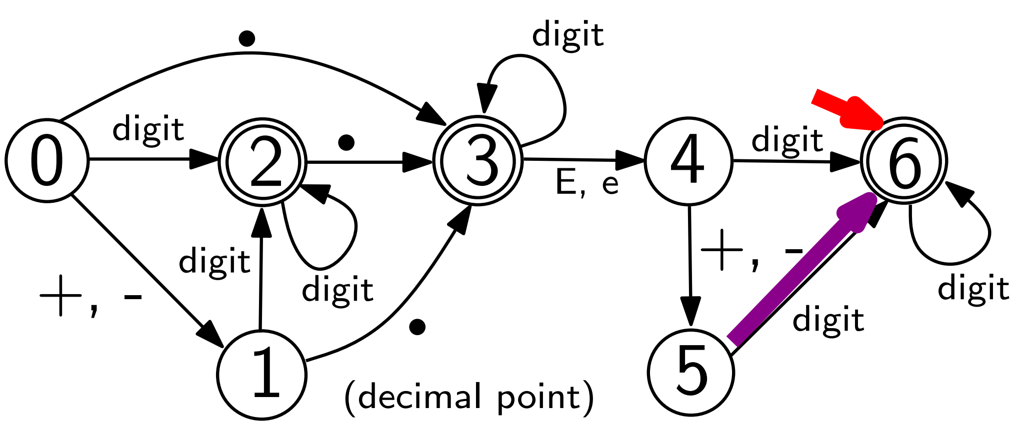


Simulation

Input: - 1 . 2 E + 3

Input "head"

Accept real constants

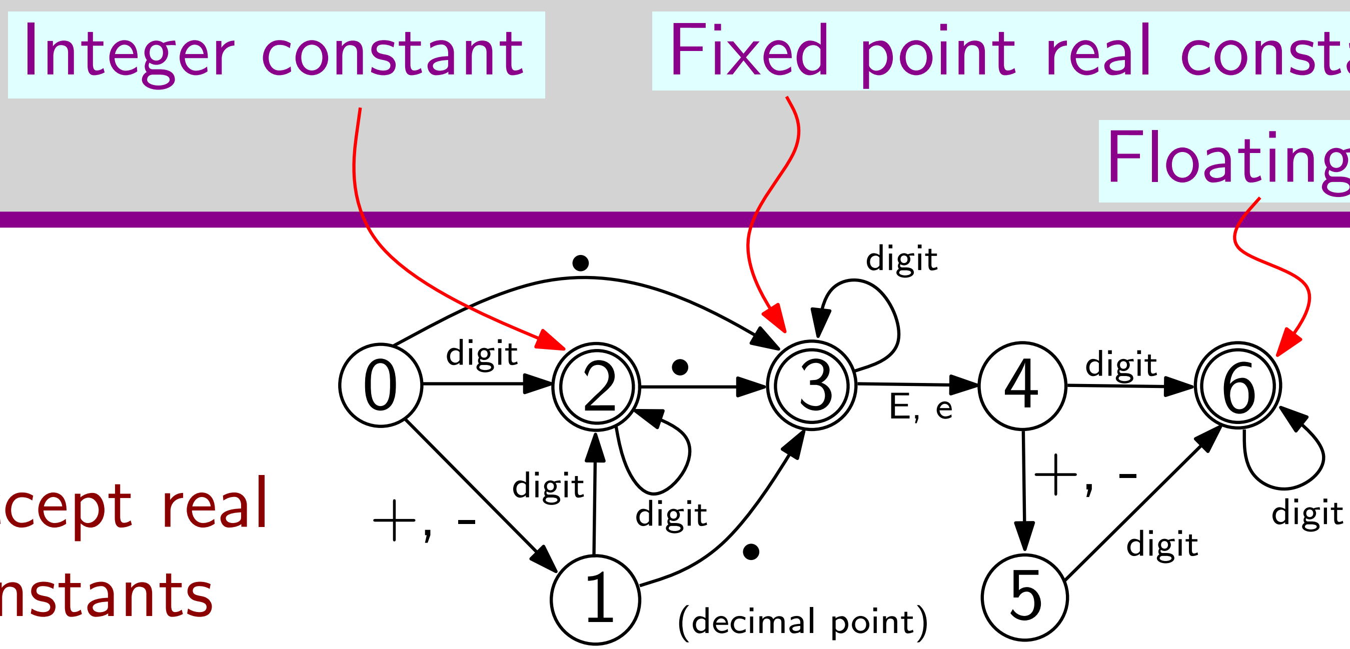


Deterministic Finite Automata

Regular languages can be accepted by *Deterministic Finite Automata*, which is the preferred way of implementing *lexical analyzers*.

Automata can be bundled together!

Attach a *lexeme type* to each final state.



Accept real constants

What have we learned

- ◇ DFAs are essentially graphs, where nodes are states, and arcs are transitions; they also have an input tape on which the input string is placed – a "head" reads one symbol at a time.
- ◇ The automaton starts in its initial state, and has it's head on the first symbol of its "input tape".
- ◇ A transition is performed only if the current state has an outgoing arrow labeled with the current symbol.
- ◇ The result of the transition is a new current state, pointed to by the current arc; the "head" advances by one symbol during the transition.
- ◇ If there is no outgoing transition for the current symbol, the automaton stops.
- ◇ If the automaton stops in a final state, the part of the string that was analyzed so far is turned into a *lexeme* with the type indicate by the final state's label.
- ◇ If the automaton does not stop in a final state, that indicates an error that stops the lexical analysis process.

Regular Expressions

- Language that defines languages (like the BNF)

- Equivalent to regular grammars and DFAs
  - For every RE E, there exists an RG G and a DFA A s.t.$\mathcal{L}(E) = \mathcal{L}(G) = \mathcal{L}(A)$. The converse is also true.

- *Definition:* Let $\Sigma = \{a, b, c, \ldots\}$ be an alphabet of symbols.
  - ϵ is an RE with $\mathcal{L}(\epsilon) = \{\epsilon\}$
  - For each $x \in \Sigma$, x is an RE with $\mathcal{L}(x) = \{x\}$
  - Given two REs r and s, $r|s$ is an RE with $\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$.
  - Given two REs r and s, $r\,s$ is an RE with $\mathcal{L}(r\,s) = \mathcal{L}(r) \cdot \mathcal{L}(s)$.
  - Given an RE r, r^* is an RE with $\mathcal{L}(r^*) = \bigcup_{i \geq 0} (\mathcal{L}(r))^i$.

Regular Expressions

- Language that defines languages (like the BNF)
- Equivalent to regular grammars and DFAs
  - For e A
    s.t. $\mathcal{L}($ ue.

$$L_1 \cdot L_2 = \{s_1\, s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2\}$$

Concatenation of languages

- *Definitio*
  symbols.
  - ϵ is a
  - For each $x \in \Sigma$, x is an RE with $\mathcal{L}(x) = \{x\}$
  - Given two REs r and s, $r|s$ is an RE with $\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$.
  - Given two REs r and s, $r\,s$ is an RE with $\mathcal{L}(r\,s) = \mathcal{L}(r) \cdot \mathcal{L}(s)$.
  - Given an RE r, r^* is an RE with $\mathcal{L}(r^*) = \bigcup_{i \geq 0} (\mathcal{L}(r))^i$.

Regular Expressions

◇ Language that defines languages (like the BNF)

◇ Equivalent to regular grammars and DFAs

– For every RE E, there exists an RG G and a DFA A

s

$$L^0 = \{\epsilon\}$$

$$L^i = \underbrace{L \cdot L \cdot \cdots \cdot L}_{i \text{ times}, \; i \geq 1}$$

◇ *De*

sym

–

–

–

$\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$.

– Given two REs r and s, $r\,s$ is an RE with $\mathcal{L}(r\,s) = \mathcal{L}(r) \cdot \mathcal{L}(s)$.

– Given an RE r, r^* is an RE with $\mathcal{L}(r^*) = \bigcup_{i \geq 0} (\mathcal{L}(r))^i$.

Regular Expressions

- ◇ Language that defines languages (like the BNF)
- ◇ Equivalent to regular grammars and DFAs
  - For every RE E, there exists an RG G and a DFA A s.t.$\mathcal{L}(E) = \mathcal{L}(G) = \mathcal{L}(A)$. The converse is also true.

$$\{s_1 \cdots s_k \mid k \geq 0, s_i \in \mathcal{L}(r), 1 \leq i \leq k\}$$

Any number of strings from $\mathcal{L}(r)$ concatenated.

- ◇ *Definition:* Let $\Sigma =$ symbols.
  - ϵ is an RE with
  - For each $x \in \Sigma$,
  - Given two REs r and s, $r|s$ is an RE with $\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$.
  - Given two REs r and s, $r\,s$ is an RE with $\mathcal{L}(r\,s) = \mathcal{L}(r) \cdot \mathcal{L}(s)$.
  - Given an RE r, r^* is an RE with $\mathcal{L}(r^*) = \boxed{\bigcup_{i \geq 0} (\mathcal{L}(r))^i}$.

Regular Expressions

Examples

$$\mathcal{L}((a|b|c)d) = \{ad, bd, cd\}$$
$$\mathcal{L}((a|b)^*) = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$$
$$\mathcal{L}((ab)^*) = \{\epsilon, ab, abab, ababab, \ldots\}$$
$$\mathcal{L}((ab)^*\, c) = \{c, abc, ababc, abababc, \ldots\}$$
$$\mathcal{L}(a\, a^*\, b) = \{ab, aab, aaab, \ldots\}$$

- *Definition:* Let $\Sigma = \{a, b, c, \ldots\}$ be an alphabet of symbols.
  - ϵ is an RE with $\mathcal{L}(\epsilon) = \{\epsilon\}$
  - For each $x \in \Sigma$, x is an RE with $\mathcal{L}(x) = \{x\}$
  - Given two REs r and s, $r|s$ is an RE with $\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$.
  - Given two REs r and s, $r\, s$ is an RE with $\mathcal{L}(r\, s) = \mathcal{L}(r) \cdot \mathcal{L}(s)$.
  - Given an RE r, r^* is an RE with $\mathcal{L}(r^*) = \bigcup_{i \geq 0} (\mathcal{L}(r))^i$.

The Language Ruby

- Developed as the pet project of Japanese programmer Yukihiro "Matz" Matsumoto.
- Multiparadigm: functional-imperative object-oriented reflective language.
- One of the few languages where regular expressions are *first-class objects*
  - can be assigned to variables
  - can be parameters to functions
  - its methods can be invoked
- Available as either interactive interpreter (command: `irb`)with *read-print-eval* loop, or as non-interactive script interpretation utility (command: `ruby`).
- Great sandbox for practicing regular expressions.
  - Watch the video for demo

Ruby Regular Expressions

<table>
<tr><td>()</td><td>the empty string</td></tr>
<tr><td>|</td><td>alternatives</td></tr>
<tr><td>[]</td><td>one-symbol alternatives</td></tr>
<tr><td>*</td><td>zero or more repetitions of the preceding</td></tr>
<tr><td>+</td><td>one or more repetitions of the preceding</td></tr>
<tr><td>?</td><td>at most one repetition of the preceding</td></tr>
<tr><td>{m,n}</td><td>at least m and at most n repetitions of the preceding</td></tr>
<tr><td>(...)</td><td>grouping</td></tr>
<tr><td>\(,\),\|,...</td><td>escaped symbols</td></tr>
<tr><td>\d</td><td>digit character, same as [0-9]</td></tr>
<tr><td>\D</td><td>non-digit character, same as [^0-9]</td></tr>
<tr><td>\w</td><td>word character, same as [0-9A-Za-z_]</td></tr>
<tr><td>\W</td><td>non-word character, same as [^0-9A-Za-z_]</td></tr>
</table>

Ruby Regular Expressions

`()`	the empty string
`|`	alternatives
`[]`	
`*`	ng
`+`	ng
`?`	at most one repetition of the preceding
`{m,n}`	at least `m` and at most `n` repetitions of the preceding
`(...)`	grouping
`\(,\),|,...`	escaped symbols
`\d`	digit character, same as `[0-9]`
`\D`	non-digit character, same as `[^0-9]`
`\w`	word character, same as `[0-9A-Za-z_]`
`\W`	non-word character, same as `[^0-9A-Za-z_]`

Range specifier

`[abc]` same as `(a|b|c)`

Ruby Regular Expressions

`()`	the empty string
`|`	alternatives
`[]`	one-symbol alternatives
`*`	zero or more repetitions of the preceding
`+`	one or more repetitions of the preceding
`?`	e preceding
`{m`	titions
`(.`	
`\(,\),|,...`	escaped symbols
`\d`	digit character, same as `[0-9]`
`\D`	non-digit character, same as `[^0-9]`
`\w`	word character, same as `[0-9A-Za-z_]`
`\W`	non-word character, same as `[^0-9A-Za-z_]`

Exclusion operator

must appear at beginning of `[...]`

Ruby Regular Expressions

()	the empty string
|	alternatives
[]	one-symbol alternatives
*	zero or
+	one or
?	at most
{m,n}	at least of the p
(...)	groupin
\(,\),\\|,...	escape
\d	digit ch
\D	non-digit character, same as [^0-9]
\w	word character, same as [0-9A-Za-z_]
\W	non-word character, same as [^0-9A-Za-z_]

Multiple ranges

Ruby's RE usage

Syntax: $/ \ldots /$

Ruby's RE usage

Syntax: / ... /

Expression

Delimiters

Ruby's RE usage

Syntax: / ... /

Expression

Delimiters

Example: `/ab*([cd]+|e?)/`

Ruby's RE usage

Syntax: / ... /

First-class value: `x = /ab*c/`

Ruby's RE usage

Syntax: / ... /

First-class value: `x = /ab*c/`

RE matching: `x.match "1abbc2"`

Ruby's RE usage

Syntax: / ... /

First-class value: `x = /ab*c/`

RE matching: `x.match "1abbc2"`

RE object

Method

String Object Argument

Ruby's RE usage

Syntax: / ... /

First-class value: `x = /ab*c/`

RE matching: `x.match "1abbc2"`

Evaluates to: `"abbc"`

Ruby's RE usage

Syntax: / ... /

First-class value: `x = /ab*c/`

RE matching: `x.match "1abbc2"`

Evaluates to: ~~`"abbc"`~~

> The result is of type `MatchData`
> Must be converted to `String` to be visualized

Ruby's RE usage

Syntax: / ... /

First-class value: `x = /ab*c/`

RE matching: `(x.match "1abbc2").to_s`

Evaluates to: `"abbc"`

The result is of type `MatchData`

Must be converted to `String` to be visualized

What Have We Learned?

- Ruby is an object oriented language
  - Expressions have the form

    `obj1.method(obj2)` (brackets optional)

    instead of

    `method(obj1,obj2)`
- Regular expressions are first class objects.
- Method `match` can be invoked to match an RE with a string.
- Result is the first occurrence of a matched substring (must be converted using `to_s`).
- REs are specified by RE-specific language.
- RE expressions must be enclosed between forward slashes.

Conclusion

- Programming languages are specified by *grammars*, in a stratified manner.
- The lower level, that of *lexical analysis*, uses *regular grammars*, and their counterpart, *regular expressions*.
  - convert a program into a sequence of *lexemes* — more in tutorial
- The higher level, called *syntactic analysis* uses more sophisticated grammars.
  - capture the structure of the language;
  - use *lexemes* as terminals
  - shall be covered in more detail next time
- *Regular expressions* are a basic data type in Ruby.
  - Ruby can be used to build a *toy lexer*.
  - Examples in the tutorial.