Parsing 2

CFGs & recursive descent parsers.

Slides by Rohit Chadha, Bill Harrison, Chris Hathhorn.

Some example CFGs: Lists-of-Numbers, "LON"

- (base) () $\in$ LON
  - i.e., the empty list is a LON
- (ind.) if n is a number and $l \in$ LON,

  then $(n, l) \in$ LON

Implicit assumption: LON is the **smallest** set satisfying these conditions.

Some example CFGs: Lists-of-Numbers, “LON”

- (base) () $\in$ LON
  - i.e., the empty list is a LON
- (ind.) if n is a number and $l \in$ LON,

  then $(n, l) \in$ LON

E.g.: LON =

`{(), (14, ()), (3, (14, ())), …}`

Some example CFGs: Lists-of-Numbers, “LON”

$LON \rightarrow ()$

$LON \rightarrow (number, LON)$

Some example CFGs: Lists-of-Numbers, "LON"

"Non-terminals"

LON → ()

LON → (*number*, *LON*)

Non-terminals appear on the left side of a production.

Some example CFGs: Lists-of-Numbers, "LON"

"Terminals"

LON → ()

LON → (*number*, *LON*)

Terminals do not appear on the left side of a production and correspond to tokens returned by the lexer.

Some example CFGs: Lists-of-Numbers, "LON"

"Productions"

LON → ()

LON → (*number*, *LON*)

Rules for forming objects are called "productions."

Some example CFGs: Numbers

number → *digit*
number → *digit number*
digit → 0
digit → 1
digit → 2
digit → 3
digit → 4
digit → 5
digit → 6
digit → 7
digit → 8
digit → 9

Some example CFGs: Numbers

number → *digit*
number → *digit number*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Some example CFGs: Regular exprs

A **regular expr** over an alphabet Σ is anything of the following form:

- $\varnothing$
- ε
- c where c is in Σ.
- A|B where A, B are REs.
- AB (or A·B) where A, B are REs.
- A* where A is a RE.

Remember this?

Some example CFGs: Regular exprs

Let RE be the smallest set such that:

- $\emptyset \in$ RE.
- $\varepsilon \in$ RE.
- $c \in$ RE, for all $c \in \Sigma$.
- If A, B $\in$ RE then $A|B \in$ RE.
- If A, B $\in$ RE then $AB \in$ RE.
- If A $\in$ RE then $A^* \in$ RE.

Some example CFGs: Regular exprs

$RE \rightarrow \varnothing$

$RE \rightarrow \varepsilon$

$RE \rightarrow c$ where $c \in \Sigma$.

$RE \rightarrow RE \mid RE$

$RE \rightarrow RE\ RE$

$RE \rightarrow RE\ *$

Some example CFGs: Regular exprs

$RE \rightarrow \emptyset$

$RE \rightarrow \varepsilon$

$RE \rightarrow c$ where $c \in \Sigma$.

$RE \rightarrow RE \mid RE$

$RE \rightarrow RE\ RE$

$RE \rightarrow RE\ ^*$

$RE \rightarrow (\ RE\ )$

Some example CFGs: Simple exprs

$$Exp \rightarrow (Exp + Exp)$$
$$Exp \rightarrow (Exp - Exp)$$
$$Exp \rightarrow (Exp * Exp)$$
$$Exp \rightarrow (Exp \backslash Exp)$$
$$Exp \rightarrow Num$$

Some example CFGs: stack machine

$SML \rightarrow$ push Num ; SML
$SML \rightarrow$ add ; SML
$SML \rightarrow$ sub ; SML
$SML \rightarrow$ mul ; SML
$SML \rightarrow$ div ; SML
$SML \rightarrow$ done

Some example CFGs: stack machine

$Stmts \rightarrow SML\ ;\ Stmts$
$Stmts \rightarrow \varepsilon$

$SML \rightarrow$ push Num
$SML \rightarrow$ add
$SML \rightarrow$ sub
$SML \rightarrow$ mul
$SML \rightarrow$ div

Some example CFGs: stack machine

"Start symbol"

Stmts → *SML* ; *Stmts*
Stmts → ε

SML → push *Num*
SML → add
SML → sub
SML → mul
SML → div

CFG for C

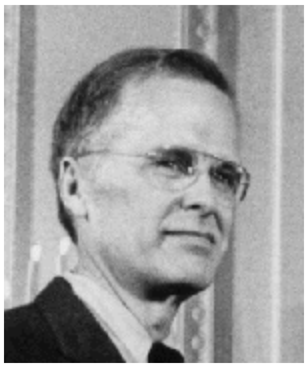
```
<selection-statement>
        ::= if ( <expr> ) <statement>
          | if ( <expr> ) <statement> else <statement>
          | switch ( <expr> ) <statement>

<iteration-statement>
        ::= while ( <expr> ) <statement>
          | do <statement> while ( <expr> ) ;
          | for ( {<expr>}? ; {<expr>}? ; {<expr>}? ) <statement>

<jump-statement> ::= goto <identifier> ;
                   | continue ;
                   | break ;
                   | return {<expr>}? ;
```

Full C Grammar is here

CFGs are also sometimes called "BNF" grammars



- "BNF" stands for "Backus-Naur Form," a common notational style for CFGs.
- John Backus (1928-2007) is one of the principal designers of FORTRAN.
- In 1954, Backus publishes "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN."
  - Backus anticipated completion of the compiler in six months. Instead, it would take two years.
  - When completed in 1956, the compiler consisted of 25,000 lines of machine code.

Ambiguous Grammars

$$Exp \rightarrow Exp + Exp$$
$$Exp \rightarrow Exp - Exp$$
$$Exp \rightarrow Exp * Exp$$
$$Exp \rightarrow Exp \backslash Exp$$
$$Exp \rightarrow Num$$

Ambiguous Grammars

$$Exp \rightarrow Exp + Exp$$
$$Exp \rightarrow Exp * Exp$$
$$Exp \rightarrow Num$$

Ambiguous Grammars

Is “1 + 2 * 3” in the language?

Exp

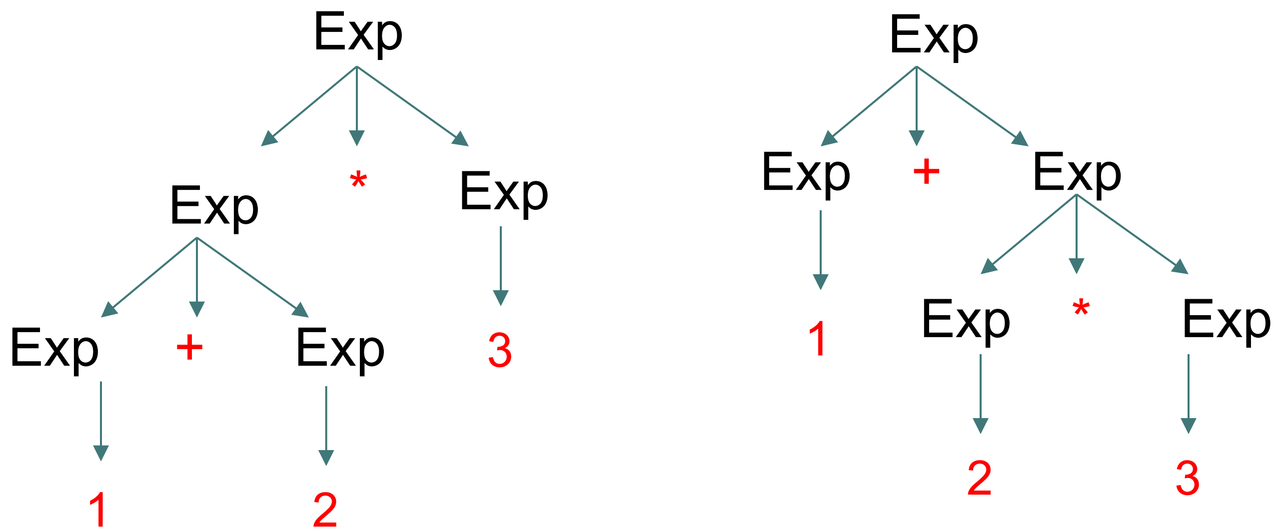
Exp

Ambiguous Grammars

Is "1 + 2 * 3" in the language?

Ambiguous Grammars

Is “1 + 2 * 3” in the language?



Ambiguous Grammars

$$Exp \rightarrow Exp + Exp$$
$$Exp \rightarrow Exp * Exp$$
$$Exp \rightarrow Num$$

Ambiguous Grammars

$$Exp \rightarrow Exp + MExp$$
$$Exp \rightarrow MExp$$

$$MExp \rightarrow MExp * MExp$$
$$MExp \rightarrow Num$$

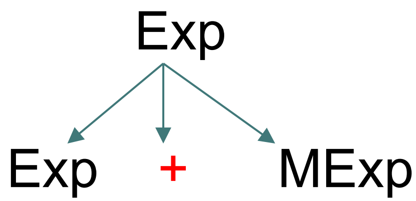
Ambiguous Grammars

Is “1 + 2 * 3” in the language?

Exp

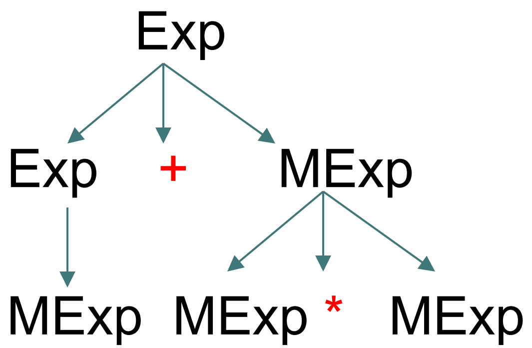
Ambiguous Grammars

Is “1 + 2 * 3” in the language?



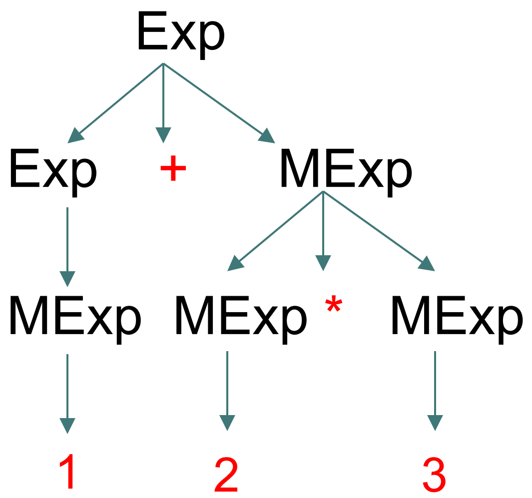
Ambiguous Grammars

Is “1 + 2 * 3” in the language?



Ambiguous Grammars

Is “1 + 2 * 3” in the language?



The Problem of Parsing

So far, we've really been talking about the sorts of languages a grammar describes.

- I.e., giving an interpretation function L : CFGs $\rightarrow$ Language, as we did for REs.

The Problem of Parsing

A *parser* accepts an input string, *s*, and a grammar, *G*, and answers the question:

"Is *s* a member of the language described by G (i.e., L(G))?"

And if the answer is "yes," a parser must also provide "proof" in the form of a derivation tree.

Recursive Descent Parsing

- Recursive descent parsers are a general kind of “top-down” parser.
- They’re very simple to understand and implement and powerful enough to handle a large class of CFGs.
- But they’re not the fastest and they can make good error messages difficult.

Recursive Descent Parsing

Is “(1 + (2 * 3))” in this lang.?

Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)

Recursive Descent Parsing

$Exp \rightarrow Num$
$Exp \rightarrow (\; Exp + Exp \;)$
$Exp \rightarrow (\; Exp - Exp \;)$
$Exp \rightarrow (\; Exp * Exp \;)$
$Exp \rightarrow (\; Exp \setminus Exp \;)$

(1 + (2 * 3))

Exp

Recursive Descent Parsing

(1 + (2 * 3))

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

$$\mathrm{Exp} \rightarrow \mathit{Num}$$

Recursive Descent Parsing

(1 + (2 * 3))

Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)

Exp

Num

Recursive Descent Parsing

(1 + (2 * 3))

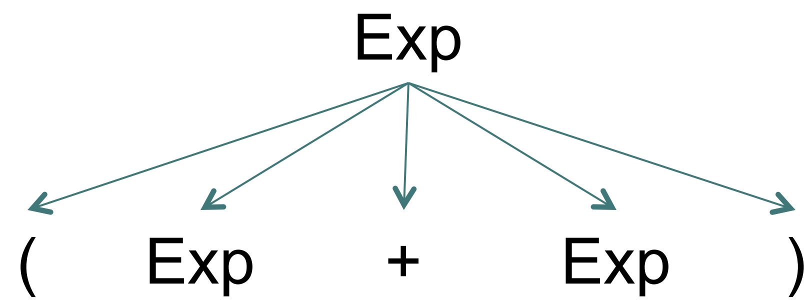
Exp → Num

Exp → (Exp + Exp)

Exp → (Exp - Exp)

Exp → (Exp * Exp)

Exp → (Exp \ Exp)



Recursive Descent Parsing

Exp → *Num*

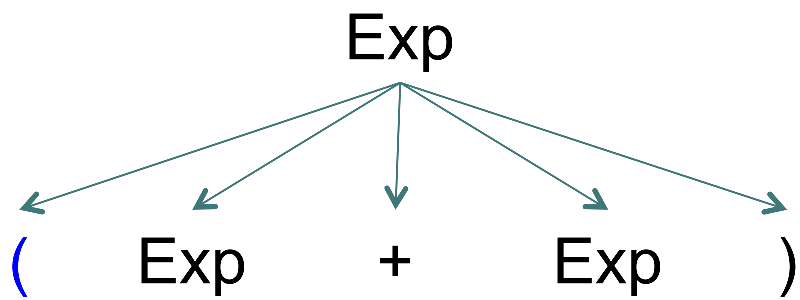
Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))



Recursive Descent Parsing

$Exp \rightarrow Num$

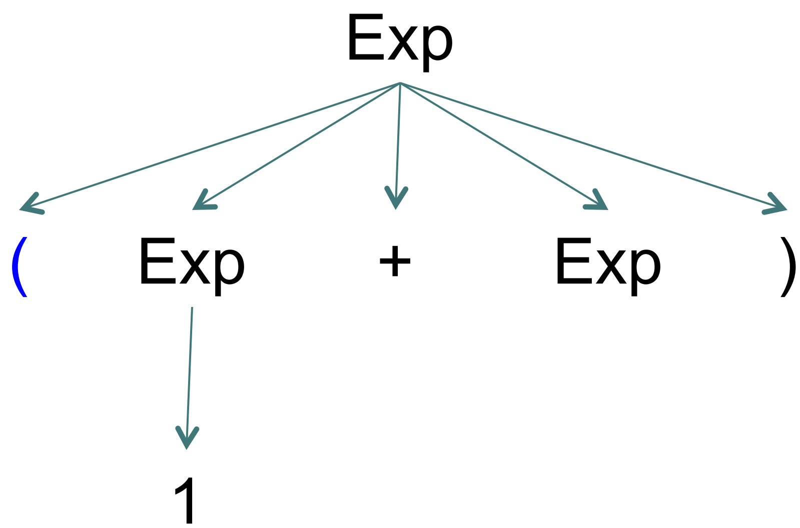
$Exp \rightarrow (\; Exp + Exp \;)$

$Exp \rightarrow (\; Exp - Exp \;)$

$Exp \rightarrow (\; Exp * Exp \;)$

$Exp \rightarrow (\; Exp \setminus Exp \;)$

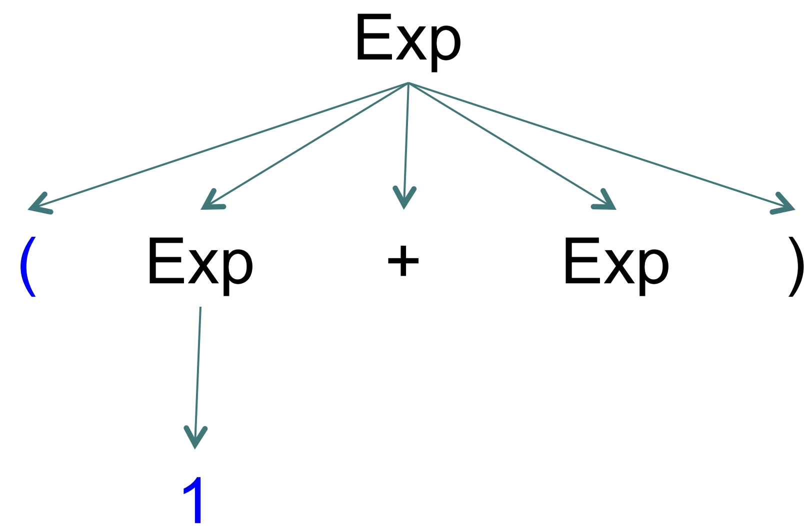
(1 + (2 * 3))



Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))



Recursive Descent Parsing

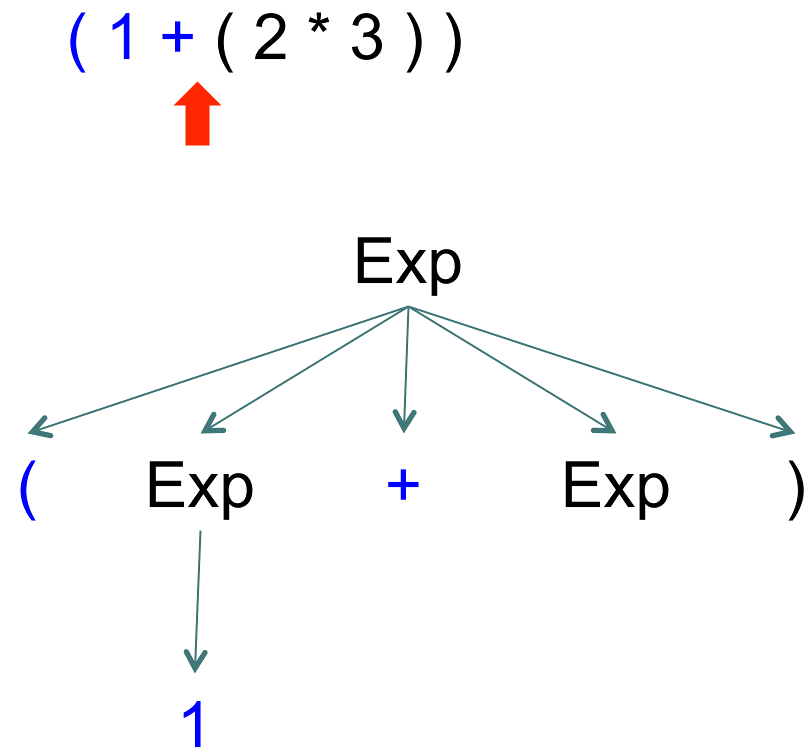
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

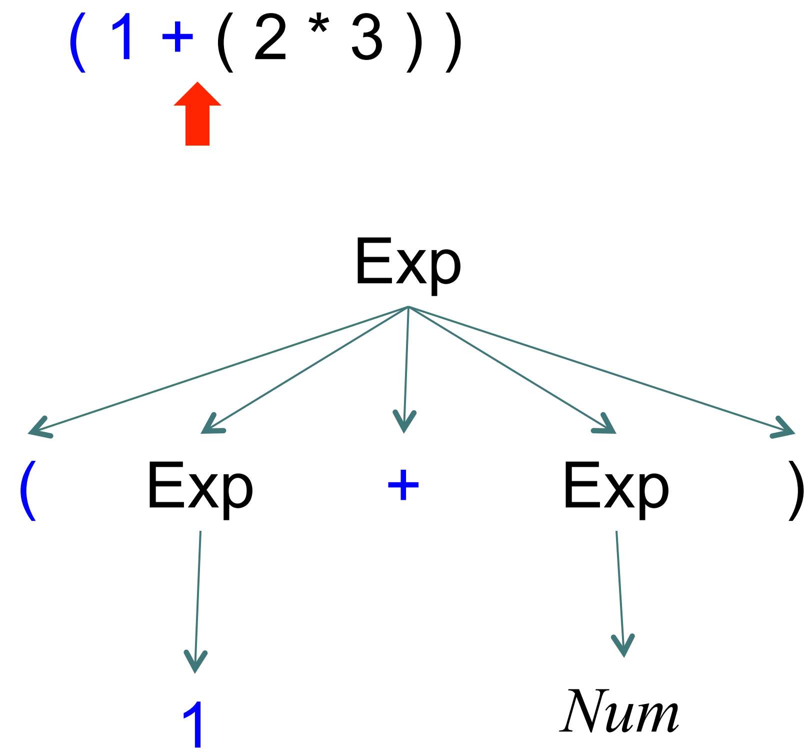
Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

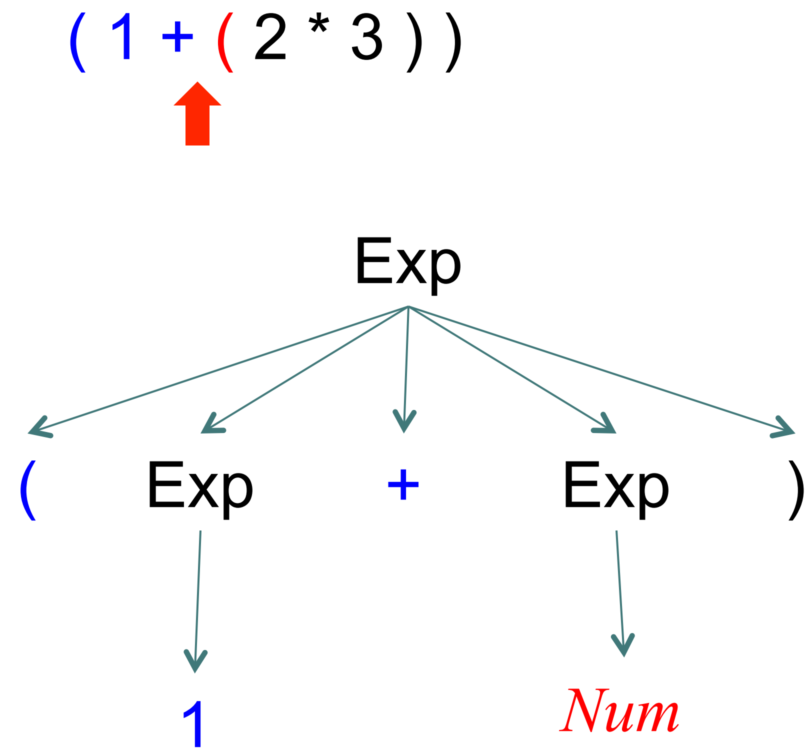
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

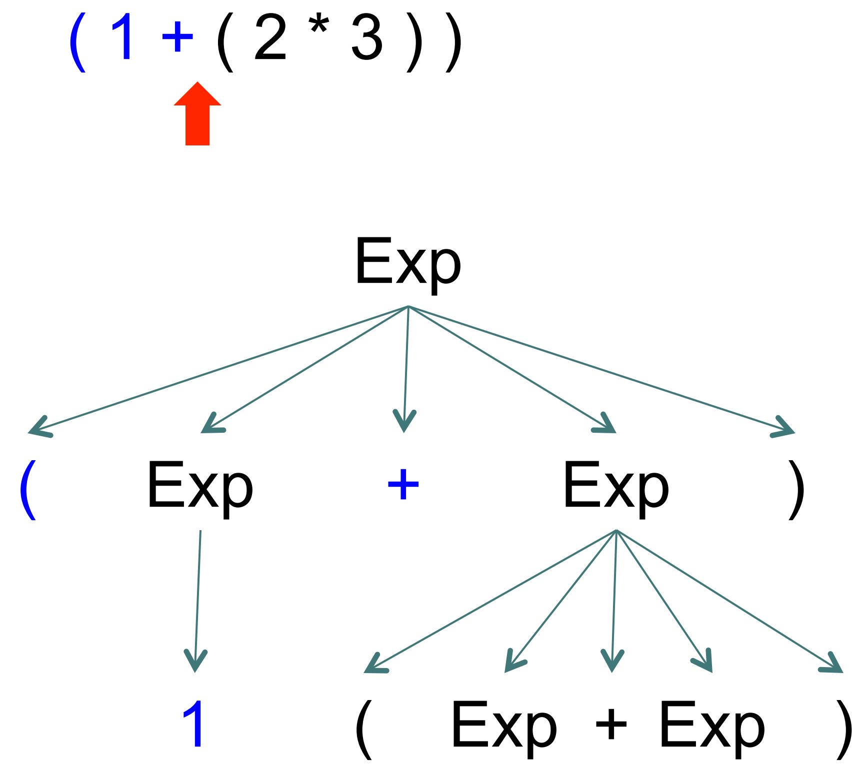
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

(1 + (2 * 3))

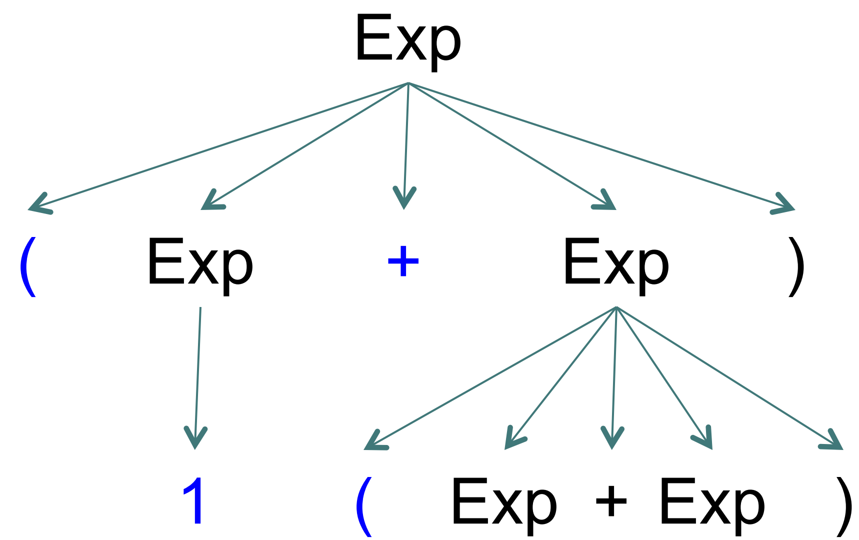
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

(1 + (2 * 3))

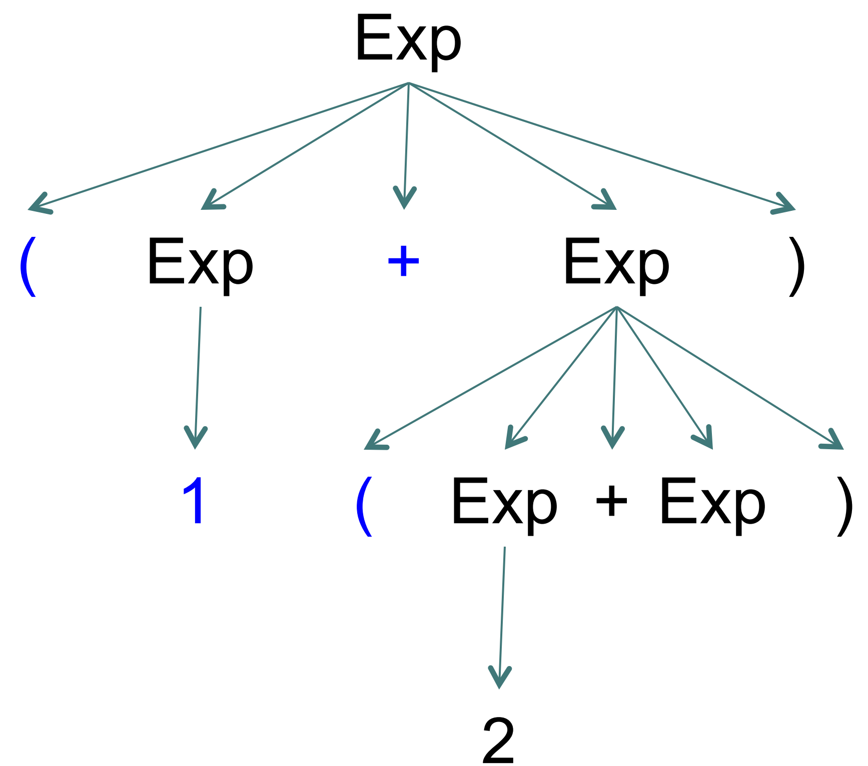
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

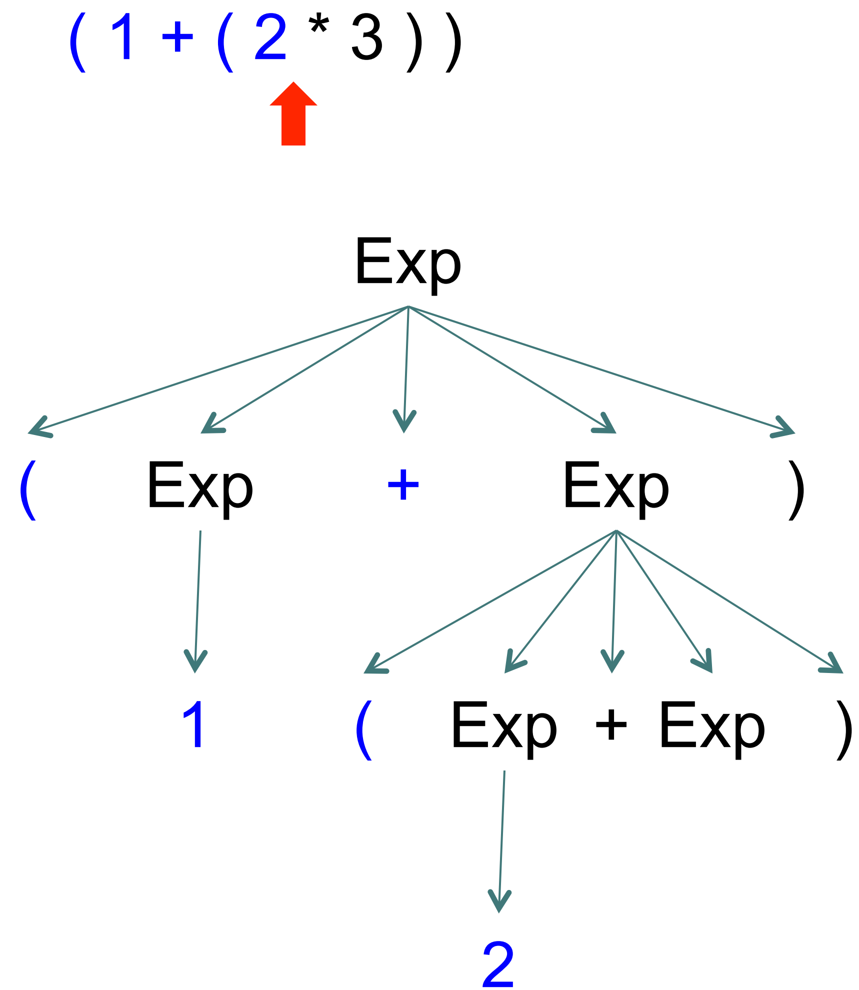
$Exp \rightarrow Num$

$Exp \rightarrow (\; Exp + Exp \;)$

$Exp \rightarrow (\; Exp - Exp \;)$

$Exp \rightarrow (\; Exp * Exp \;)$

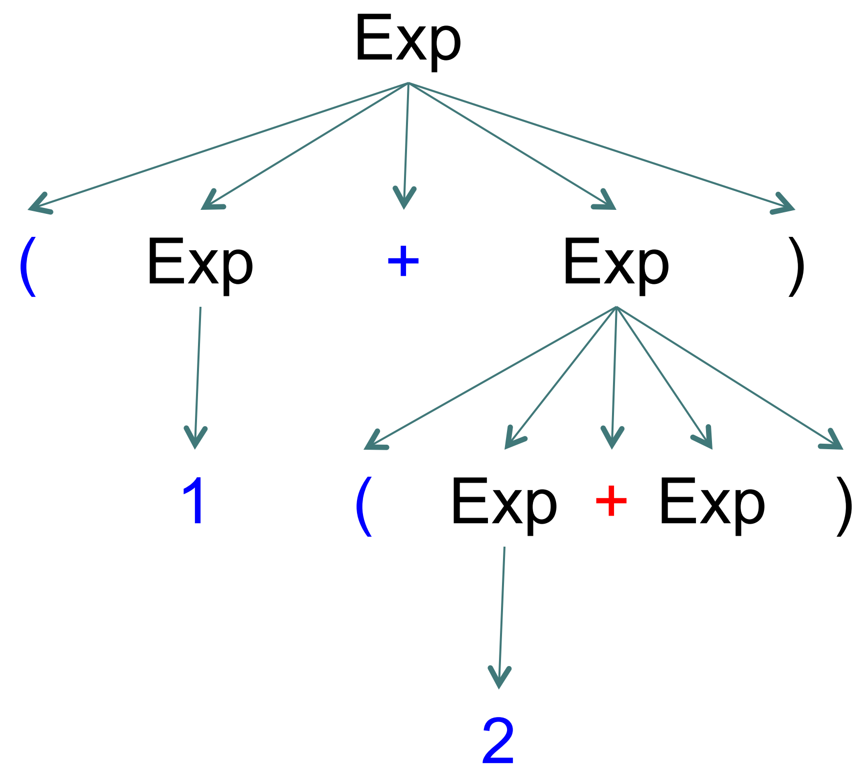
$Exp \rightarrow (\; Exp \setminus Exp \;)$



Recursive Descent Parsing

(1 + (2 * 3))

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

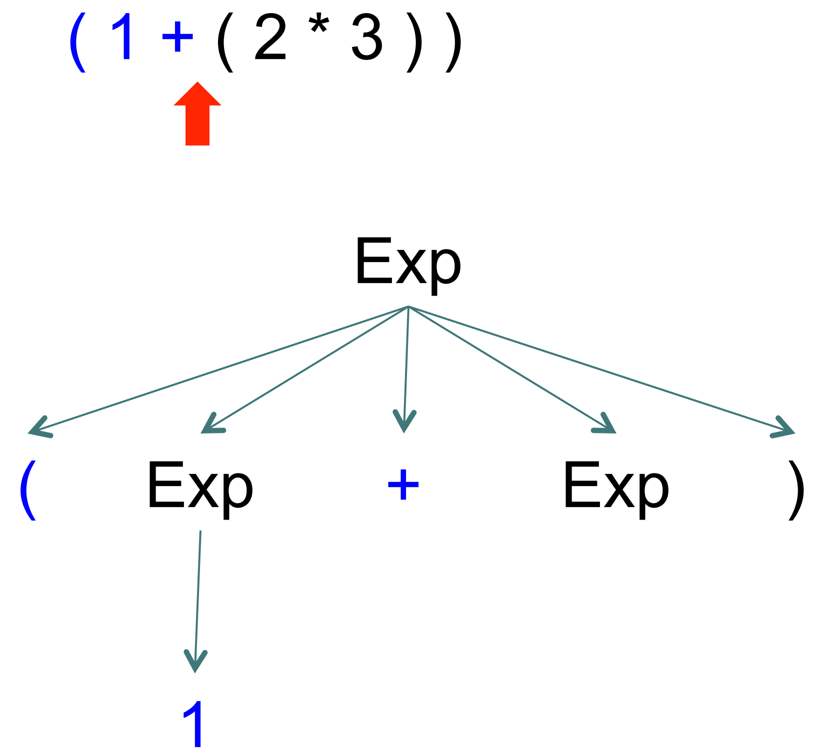
$Exp \rightarrow Num$

$Exp \rightarrow (\; Exp + Exp \;)$

$Exp \rightarrow (\; Exp - Exp \;)$

$Exp \rightarrow (\; Exp * Exp \;)$

$Exp \rightarrow (\; Exp \setminus Exp \;)$



Recursive Descent Parsing

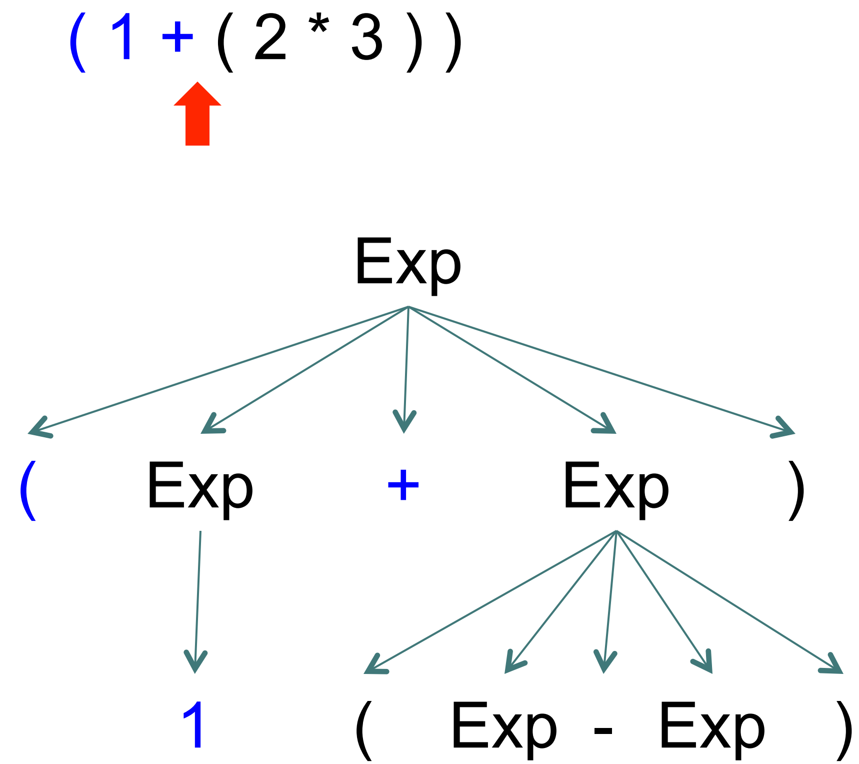
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

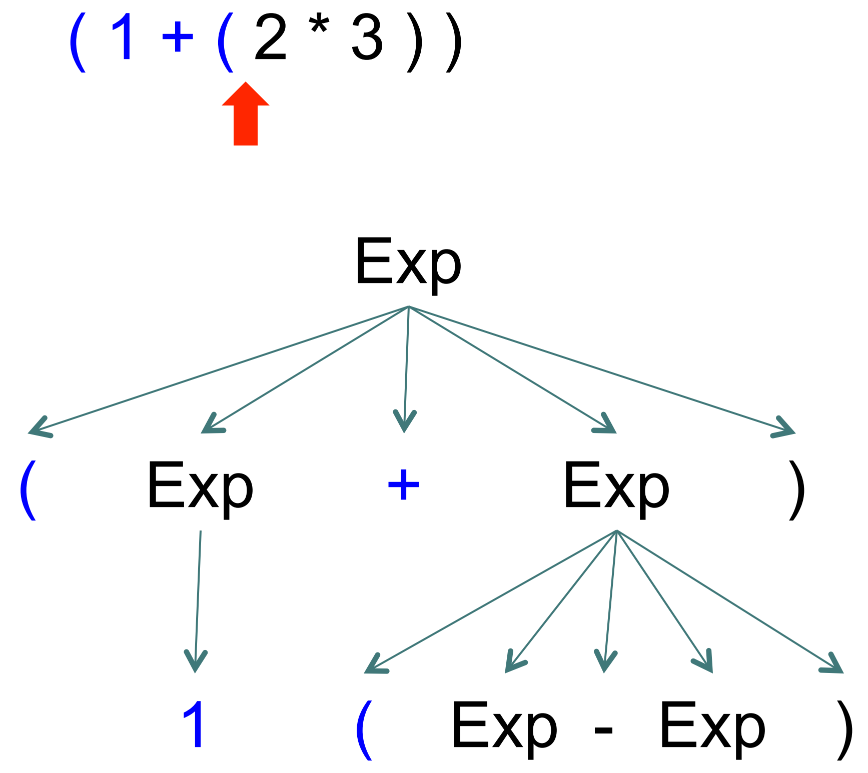
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

Exp → *Num*

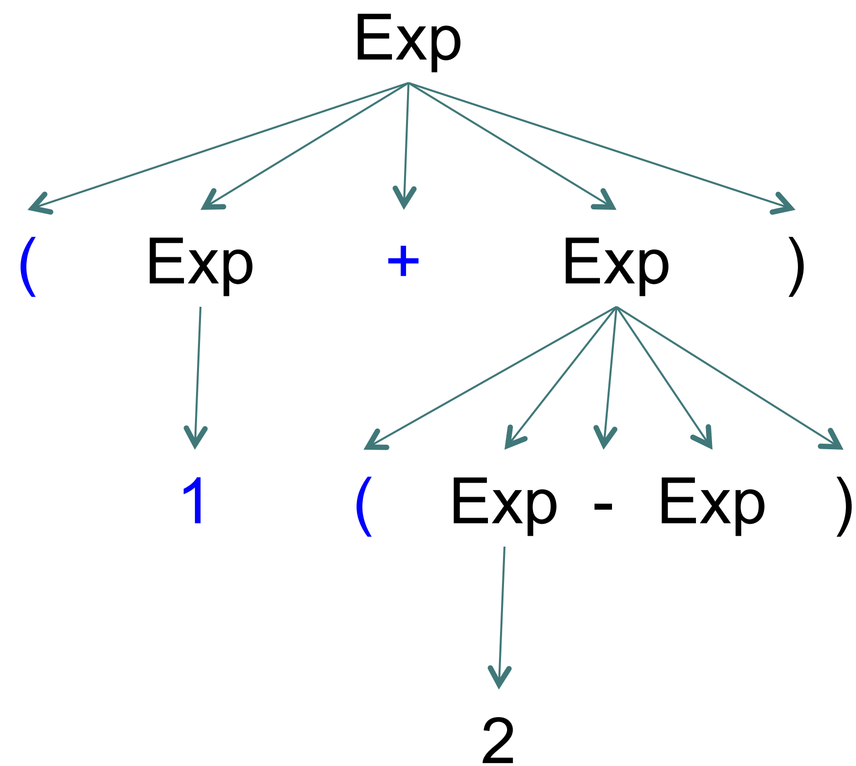
Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))



Recursive Descent Parsing

$Exp \rightarrow Num$

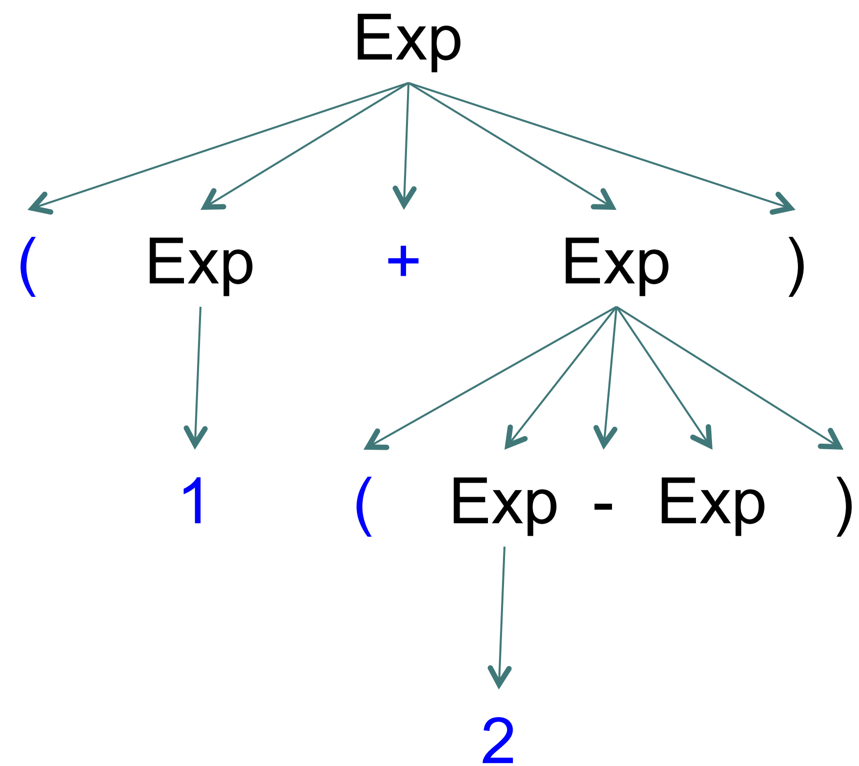
$Exp \rightarrow (\; Exp + Exp \;)$

$Exp \rightarrow (\; Exp - Exp \;)$

$Exp \rightarrow (\; Exp * Exp \;)$

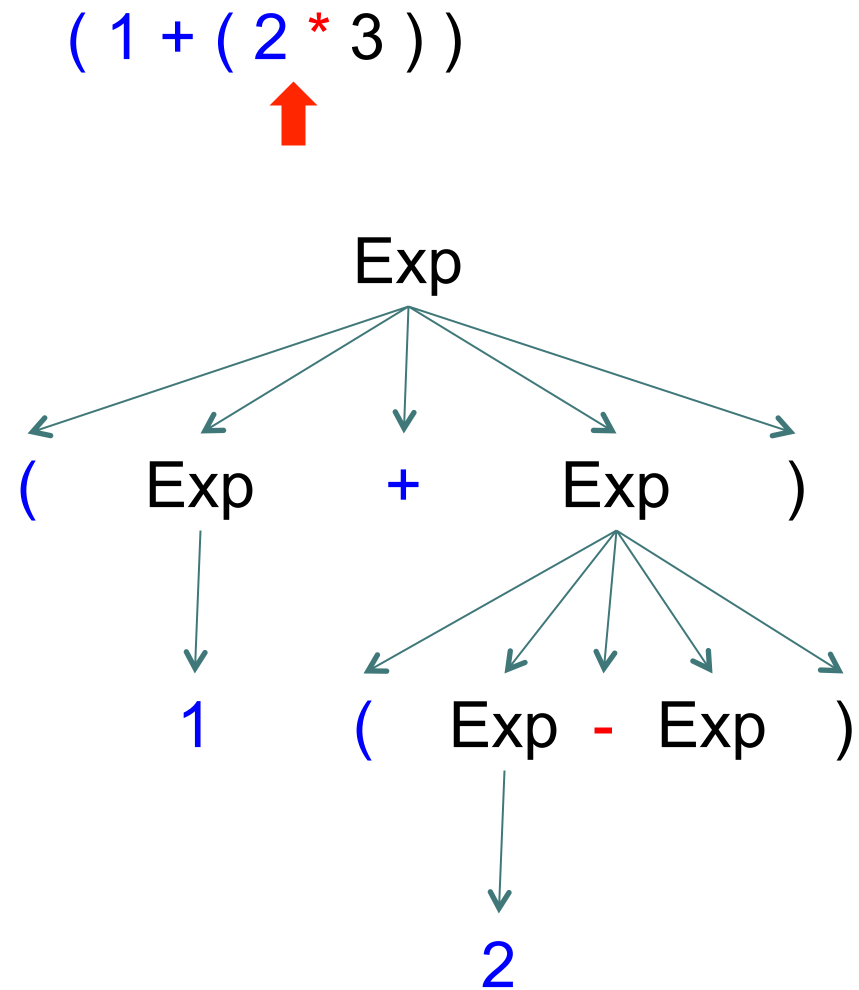
$Exp \rightarrow (\; Exp \backslash Exp \;)$

(1 + (2 * 3))



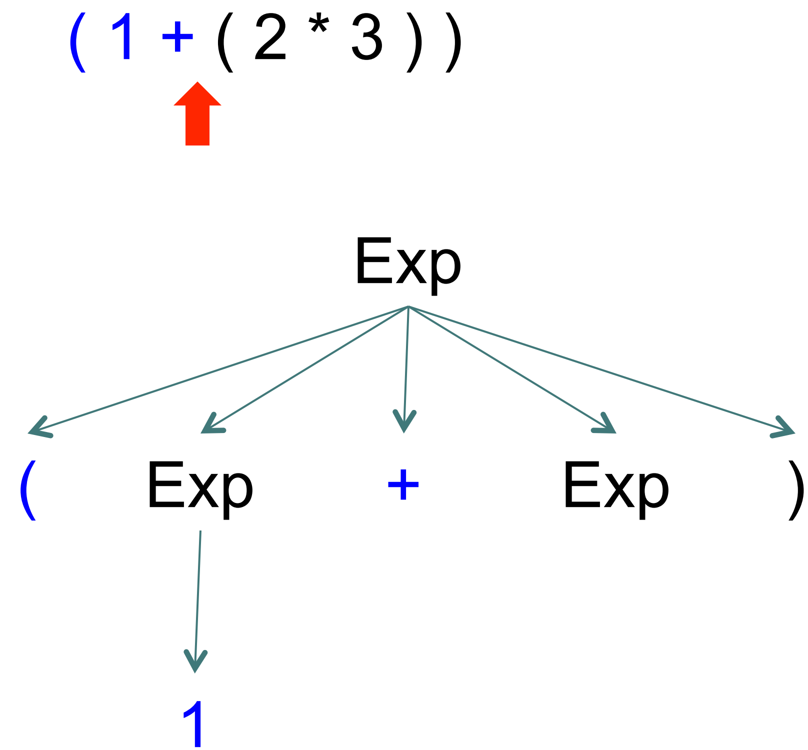
Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



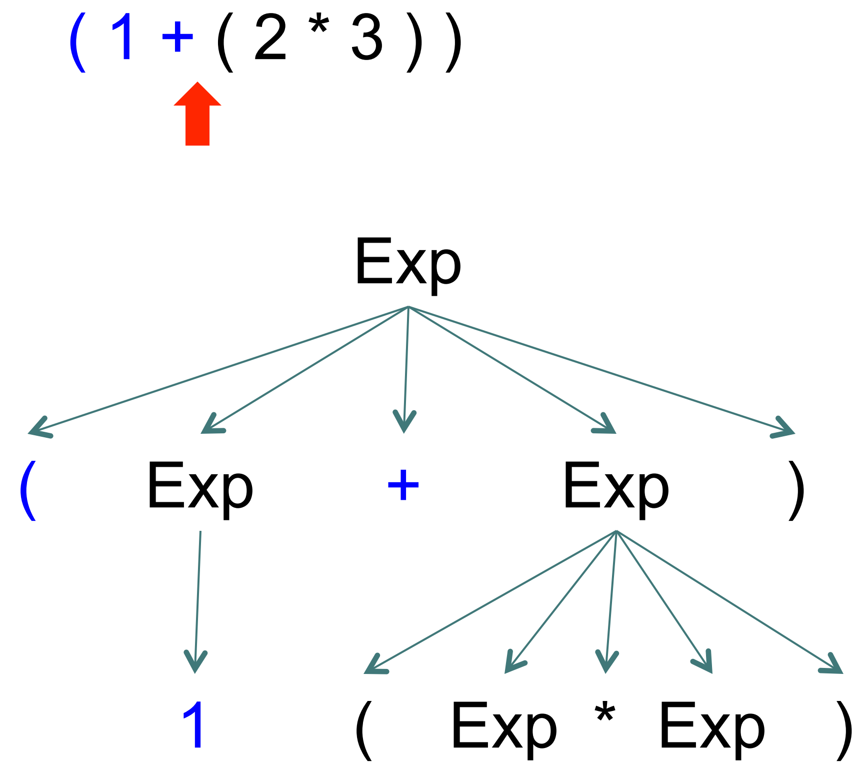
Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



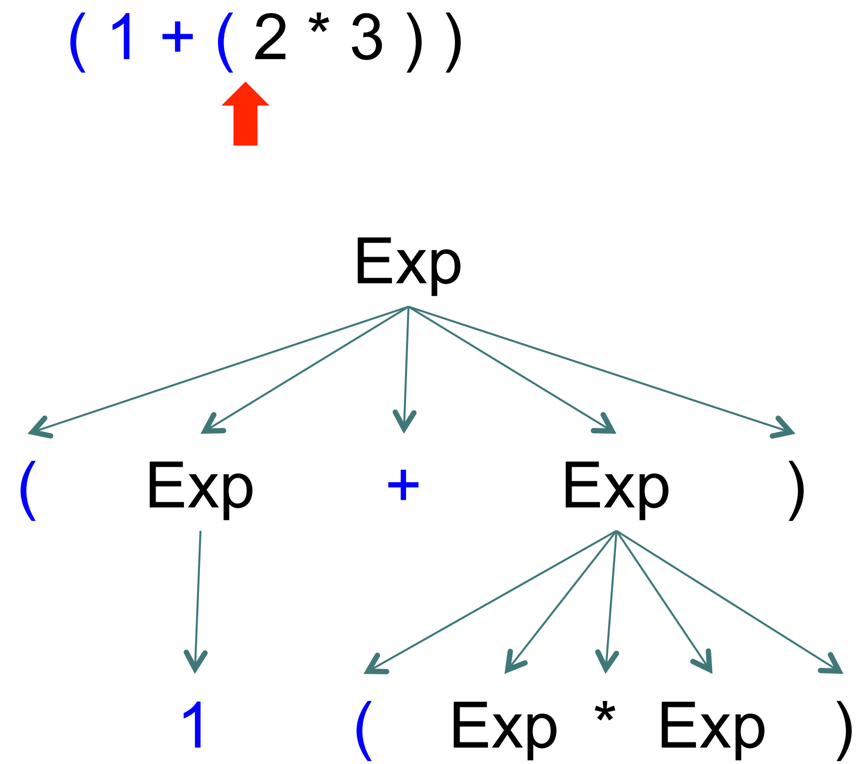
Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



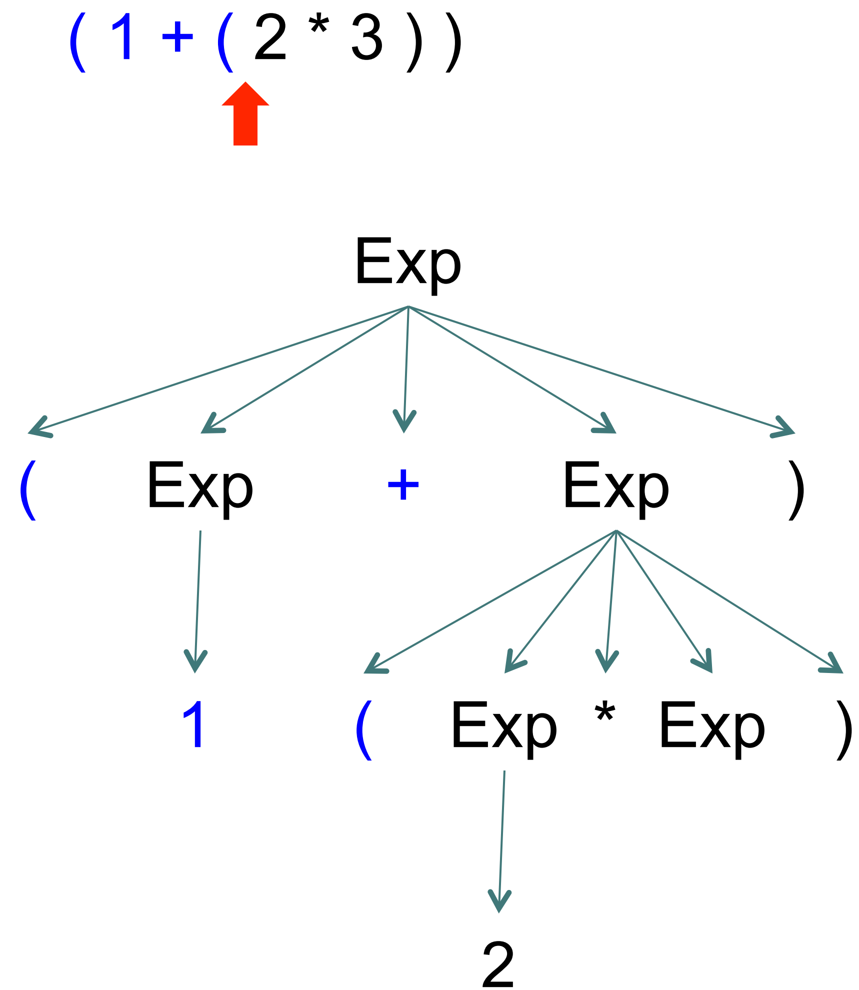
Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

(1 + (2 * 3))

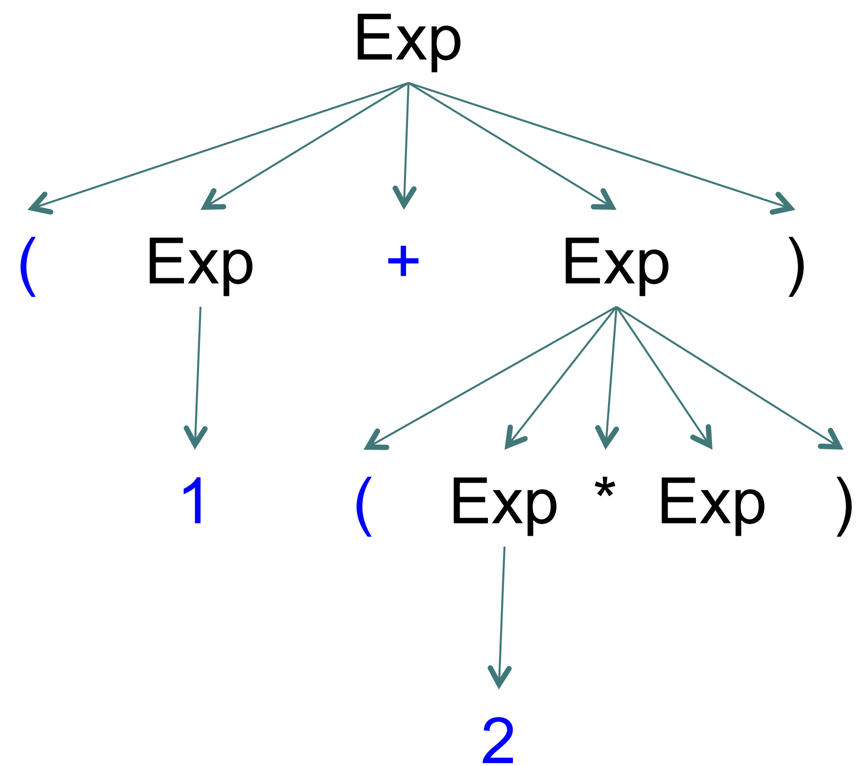
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

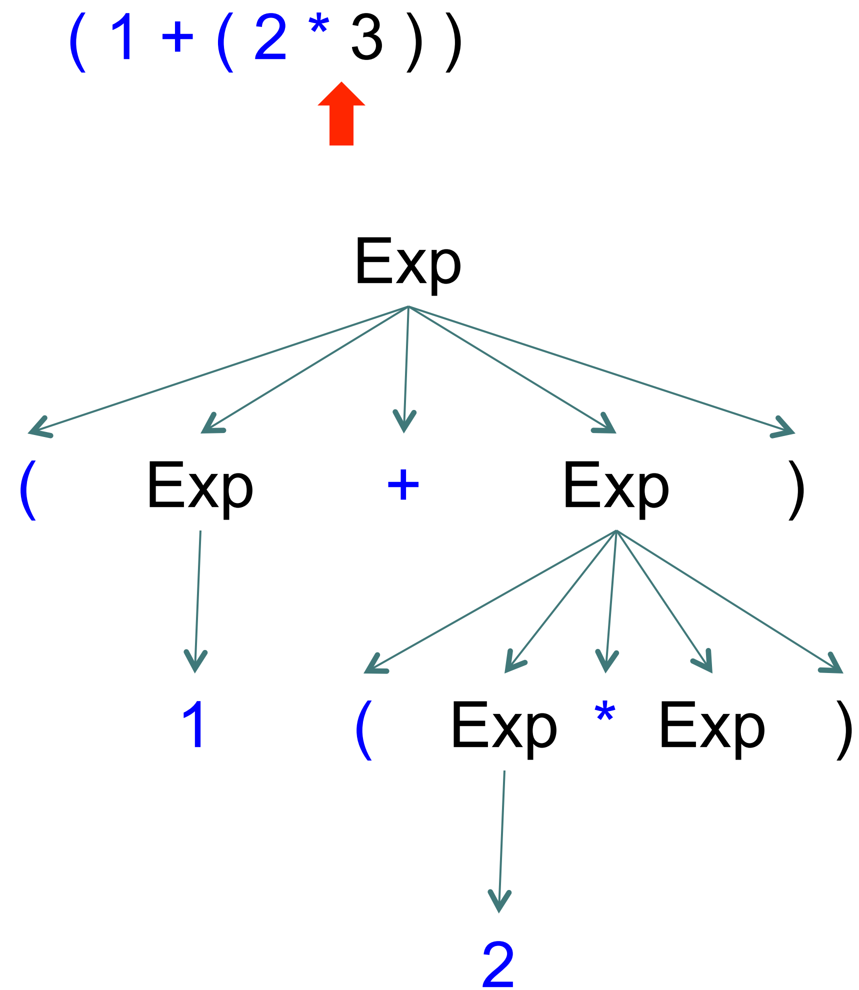
Exp → *Num*

Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

Exp → *Num*

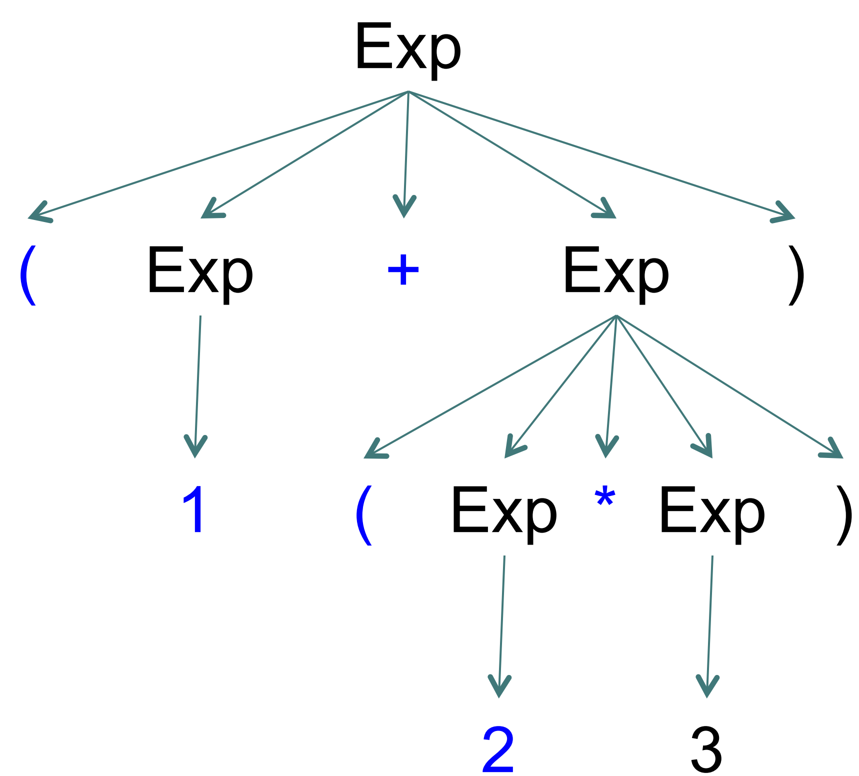
Exp → (*Exp* + *Exp*)

Exp → (*Exp* - *Exp*)

Exp → (*Exp* * *Exp*)

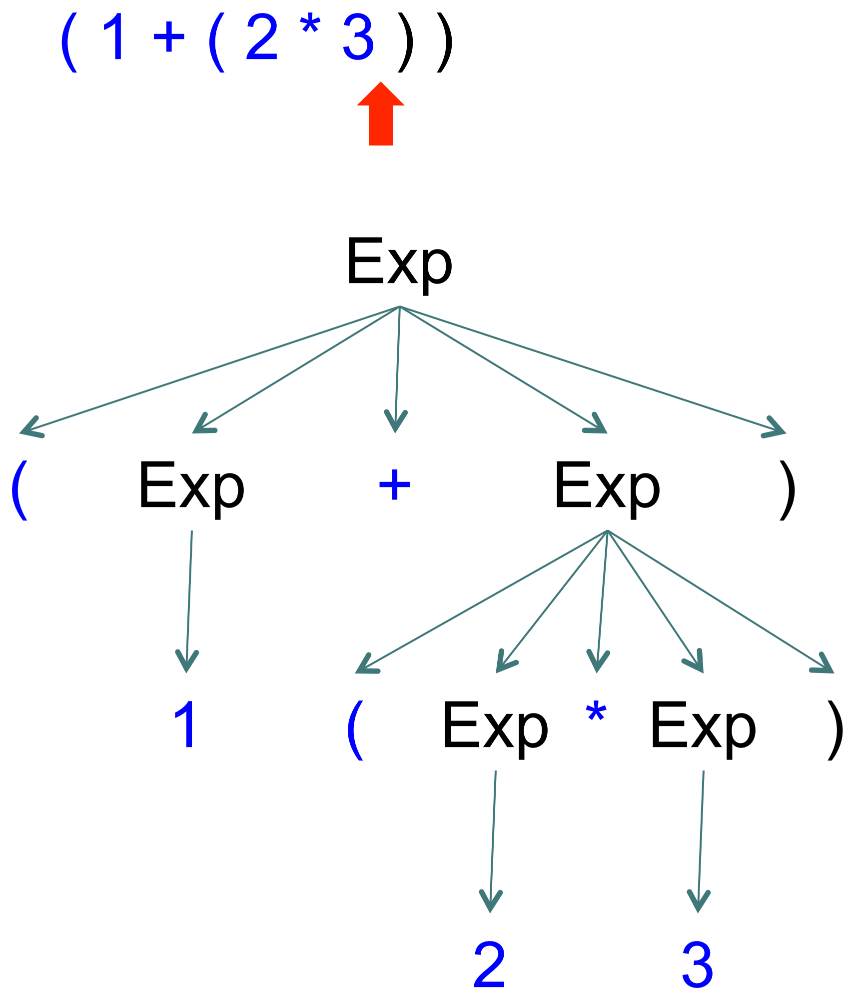
Exp → (*Exp* \ *Exp*)

(1 + (2 * 3))



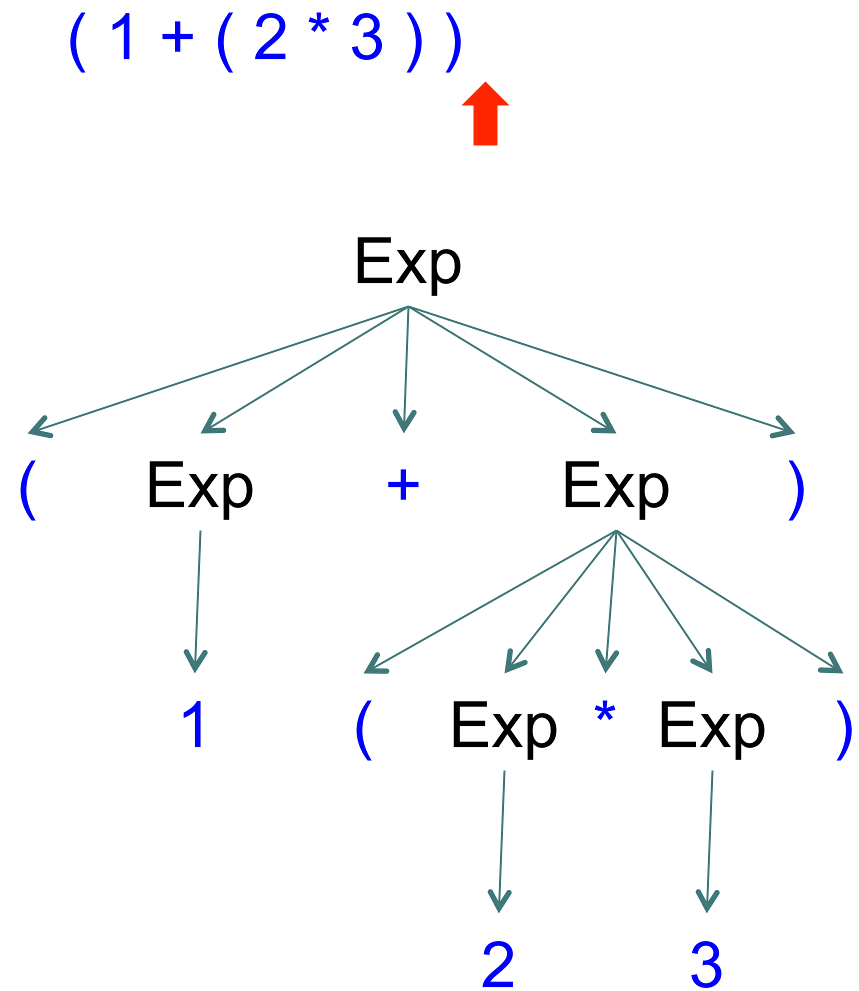
Recursive Descent Parsing

Exp → *Num*
Exp → (*Exp* + *Exp*)
Exp → (*Exp* - *Exp*)
Exp → (*Exp* * *Exp*)
Exp → (*Exp* \ *Exp*)



Recursive Descent Parsing

$Exp \rightarrow Num$
$Exp \rightarrow (Exp + Exp)$
$Exp \rightarrow (Exp - Exp)$
$Exp \rightarrow (Exp * Exp)$
$Exp \rightarrow (Exp \setminus Exp)$



But what if no parens?

Exp → *Num*
Exp → *Exp* + *Exp*
Exp → *Exp* - *Exp*
Exp → *Exp* * *Exp*
Exp → *Exp* \ *Exp*

Parsing summary so far…

- A **Context-free grammar** describes a language.
  - Terminals: tokens from the lexer.
  - Non-terminals: have productions (rules) for deriving our language.
  - One non-terminal is designated the “start symbol,” the root of all derivations.
- Parsing is performed by repeatedly choosing which production to use next.
  - There are several ways of choosing which production to use next.
- Grammars can be ambiguous, i.e., admit several valid parses.
  - Can often rework grammar & remove ambiguity, but not always.
  - Typically must choose one of many possible parse trees