Homework & Announcements

- ☐ Reading: Chapter
- ☐ Homework: Exercises at end
- ☐ Due: 11/5

COS 140: Foundations of Computer Science

Specifying Programming Languages:
Backus–Naur Form (BNF)

Fall 2018

Problem

- Problem: how to *specify* a programming language?
- Have to have a way to describe its syntax (and, possibly, its semantics)

Syntax and Semantics

- *Syntax*:
  - Form of something (e.g., language)
  - Describes relationships between components
- *Semantics:*
  - Meaning of something
  - Describes what the statements will do
- Need to capture formally so it's clear how language is to be used.

Syntax Formalisms

- What is needed in a syntax formalism?
  - Able to specify language's grammar for users and compiler designers
  - Easy to write and understand
  - Expressive enough to capture all programming languages
  - Easy to translate into algorithms for machine translation of language

Syntax Formalisms

- Formalisms often expressed as *production rules*:
  - $LHS \rightarrow RHS$
  - Match LHS with what you have, produce RHS
  - E.g.: S $\rightarrow$ NP VP
- As we'll see: can also think of going backwards
  - If you have NP and VP $\Rightarrow$ have a valid sentence S.

Backus–Naur Form

- BNF is the most common way of specifying *grammar* of a programming language
- Also important for:
  - Computability theory
  - Natural language processing
  - Many other places in CS where you need to specify a grammar

Language Terminology

- *String:* any combination of characters in the language – may be the whole program
- *Lexemes:* lowest-level unit of language – analogous to words in English
- *Syntactic category:* classes of lexemes that play the same role in the structure of a statement – analogous to parts of speech
- *Tokens:* Low-level syntactic categories corresponding to the lexemes
- *Constituents:* tokens or groups of tokens that are put together in ways specified by the grammar – analogous to noun phrases, etc.
- *Grammar:* specification of the syntax; a set of rules describing the legal strings of the language
- *Terminal* and non-terminal symbols

Types of Languages: The Chomsky Hierarchy

- As go down the hierarchy, languages increase in complexity, more machinery needed to recognize them
- *Regular languages* (regular expressions) – tokens
  - Single symbol on the LHS; RHS has at most one non-terminal:
  - E.g.: S → aS | b – generates a^+b
- *Context-free languages* – programming languages
  - LHS: single symbol; terminals, non-terminals in RHS
  - E.g.: S → aSb | nil – generates a^nb^n
- *Context-sensitive languages*
  - RHS has at least as many symbols as LHS
  - E.g.: aSc → aSbSc
- *Recursively-enumerable languages* – Turing-equivalent representation
  - No restrictions on LHS, RHS
  - E.g.: aaaS → aaaTQ

Backus-Naur Form (BNF)

- Grammar formalism for context-free languages (only one symbol on LHS)
- Terminology:
  - *Rewrite (production) rules* – rules that rewrite current pattern into a new one
  - *LHS, RHS* – parts of rule: LHS $\rightarrow$ RHS
  - *Terminal symbols* – symbols that appear only on RHS – i.e., do not get replaced by anything
  - *Non-terminal symbols* – symbols that appear in some LHS
  - *Alternatives* (|) – different RHS's that can be used with the LHS

Backus-Naur Form (BNF)

- E.g.: <expr>→<var> | <var> + <var>
  <var>→A | B | C
- Often written with ::= instead of →:
  <var> ::= A | B | C

Example: Producing a String from BNF

$$\langle\text{expr}\rangle \rightarrow \langle\text{var}\rangle \mid \langle\text{var}\rangle + \langle\text{var}\rangle$$
$$\langle\text{var}\rangle \rightarrow \text{A} \mid \text{B} \mid \text{C}$$

- Start with the kind of constituent you want to produce

  <expr>
- Replace each non-terminal in LHS with a RHS that appears in the rule for which that non-terminal is in the LHS

  <var> + <var>
- Keep doing this until there are only non-terminals in the string

  A + <var>⇒ . . . ⇒A + B

Derivation

- *Start symbol:* where derivations begin to derive all possible strings (programs)
- *Sentential form:* any string derived from the start symbols using the rewrite rules
- *Derivation:* rewrite sentential forms until have all terminal symbols
- Options:
  - Order of rewriting – left-most, right-most derivations
  - Alternative used for rewriting

Recursion

- In BNF: LHS appears in the RHS
- Allows infinite language from finite grammar
- Cannot specify the number of times rule will be applied
- E.g.: to allow any number of additions in an expression:

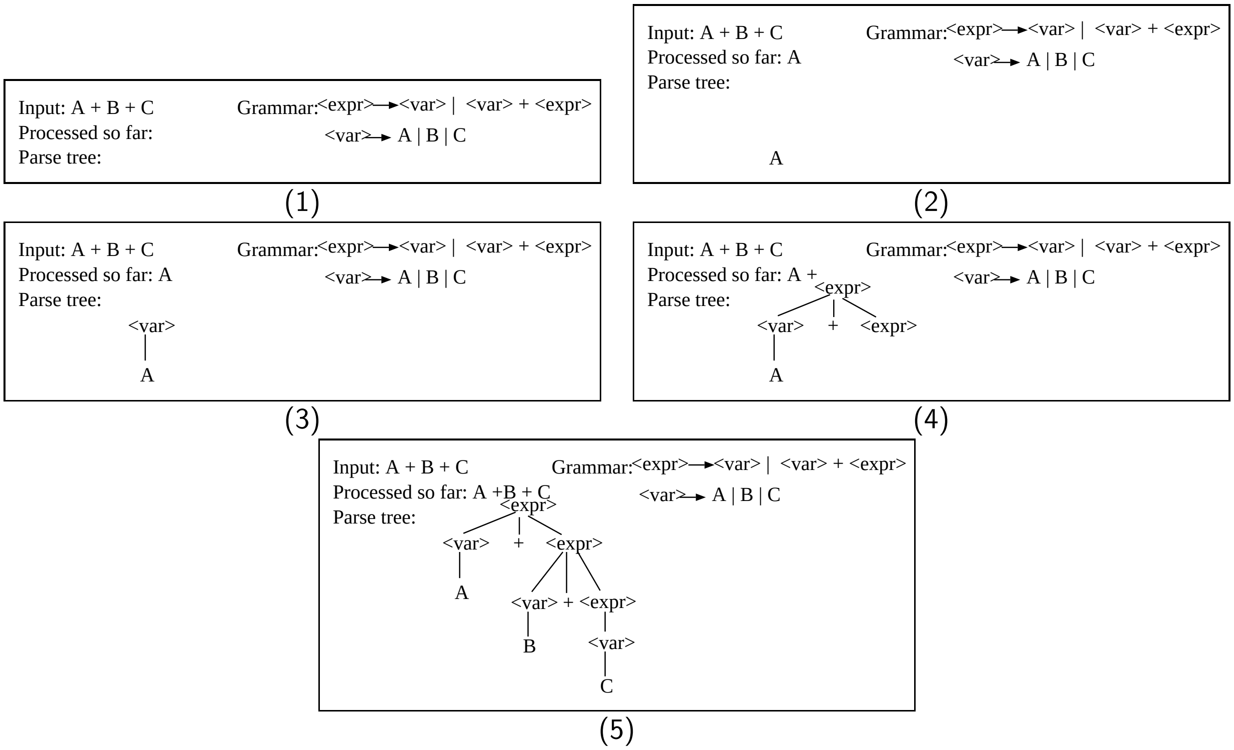
  <expr>→<var> | <var> + <expr>
- Need an alternative that can stop the recursion!

Parsing

Parsing

- What is *parsing*?
  - Checking the legality of input against a grammar
  - Producing a *parse tree* that shows the relationships between the tokens
  - Parse trees record derivations
- Start with a string of only tokens (terminal symbols)
- Find a RHS pattern in the string, replace with the LHS.
- Continue until you have the start symbol
- If you can do this: the input was a valid sentence in the language
- Create a parse tree by showing how we replace terminal/non-terminal symbols using appropriate rules

Parsing Example



Ambiguity

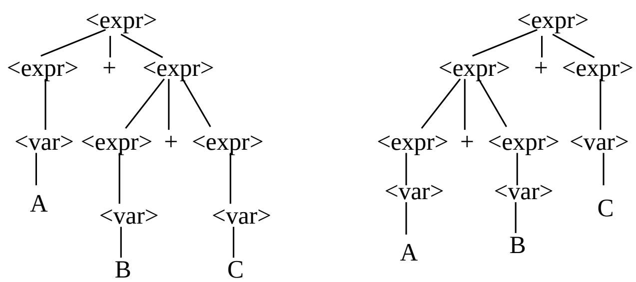
- More than one different, legal, correct parse trees for the same string
- Problem, since parse tree indicates relationship between the constituents
- Property of the grammar

Ambiguity Example

- Grammar:

  <expr>→<var> | <expr> + <expr>

  <var>→A | B | C
- Parse trees:

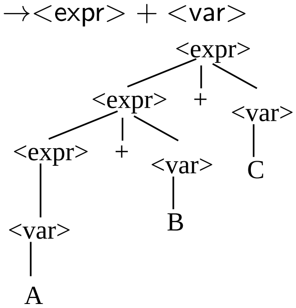


- Sub-expressions that are more deeply nested in the tree are evaluated first

An Unambiguous Grammar for Addition Expressions

- Need to ensure that the expression can be evaluated in only one way
- Want the expression to be evaluated left to right ⇒ first expression formed (lowest in tree) needs to be to the left
- Replace rule in previous with:

<expr>→<expr> + <var>



Operator Precedence

- Need a constituent for each set of operators with the same precedence
- Operators with the highest precedence need to be built at the lowest level in the tree
- Get left associativity if we parse left → right, keep recursion to the left

  <expr>→<expr> + <var>
- Right associativity: recursion to the left.

Levels of Precendence

- Items in parentheses and identifiers, numbers: *factor*

  A, B, (C + D)
- Multiplication and division: *term*

  A*B, A/B
- Addition and subtraction: *expression*

  A+B, A-B

Example Grammar

<expr>→<expr> + <term> |
<expr> − <term> |
<term>
<term>→<term> ∗ <factor> |
<term> / <factor> |
<factor>
<factor>→ (<expr>) | <id>
<id>→ A | B | C | D

Parsing

- Can think of there being a "state" of the parse
  - Records where we are in the process
  - E.g.: <expr> + <term> would mean that we've turned the tokens we've read so far into the non-terminal <expr>, the terminal symbol (token) +, and the non-terminal <term>
- Idea of state can be used to parse using a *state machine* or automata – we'll talk about this later in course
- Can also think of there being some tokens that have not yet been read
- Start with an empty state, and with all terminals unread

Parse of A + B * C - D

State:
(empty)
Input left: A + B * C - D

No grammar rules apply to empty state.
Read next token (A).

A + B * C - D

State:
A
Input left: + B * C - D

Only rule that applies is:
<id> ::= A

A + B * C - D

State:
<id>
Input left: + B * C - D

Only rule that applies is:
<factor> ::= <id>

<id>
A + B * C - D

State:
<factor>
Input left: + B * C - D

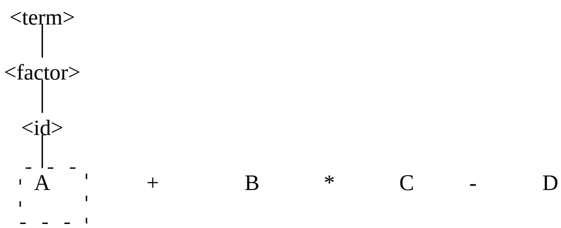
Only rule that applies is:
<term> ::= <factor>

<factor>
<id>
A + B * C - D

Parse of A + B * C - D

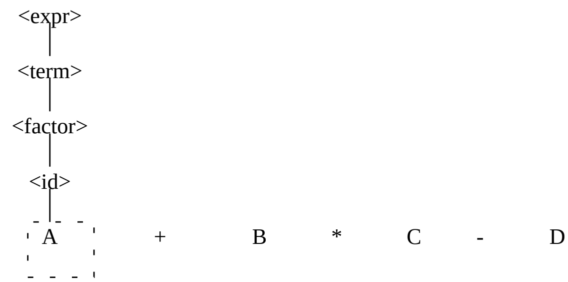
State:
<term>
Input left: + B * C - D

Only rule that applies is:
<expr> ::= <term>



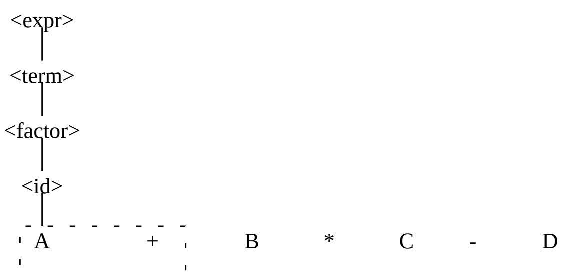
State:
<expr>
Input left: + B * C - D

Although <expr> is the current state, there are still input tokens left, so we're not done.
Read next token (+).



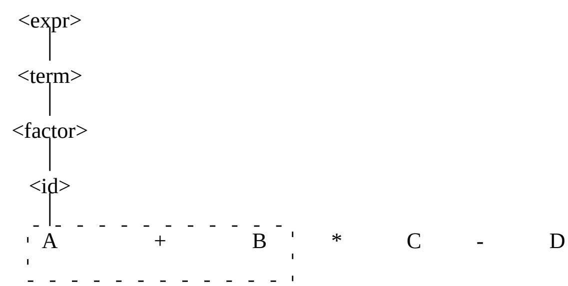
State:
<expr> +
Input left: B * C - D

No rule applies.
Read next token (B).



State:
<expr> + B
Input left: * C - D

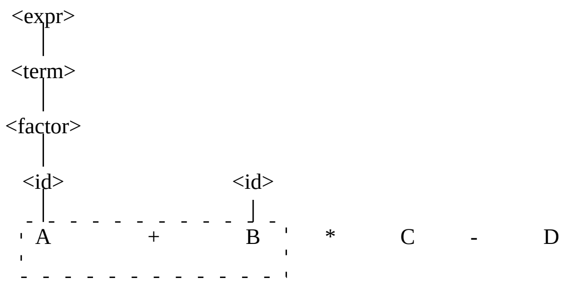
Can't replace whole thing.
Use <id> ::= B



Parse of A + B * C - D

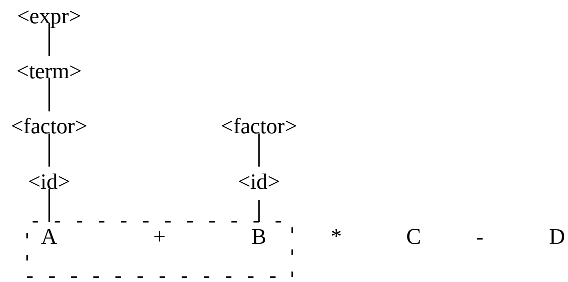
State:
<expr> + <id>
Input left: * C - D

Can't replace whole thing.
Use <factor> ::=<id>



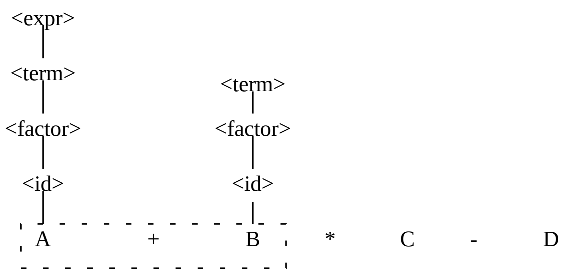
State:
<expr> + <factor>
Input left: * C - D

Can't replace whole thing.
Use <term> ::= <factor>



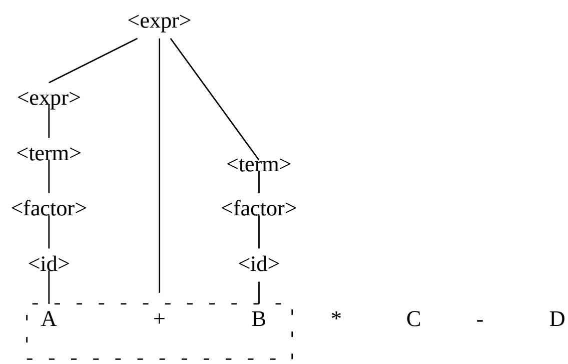
State:
<expr> + <term>
Input left: * C - D

Can apply <expr> ::= <expr> + <term>



State:
<expr>
Input left: * C - D

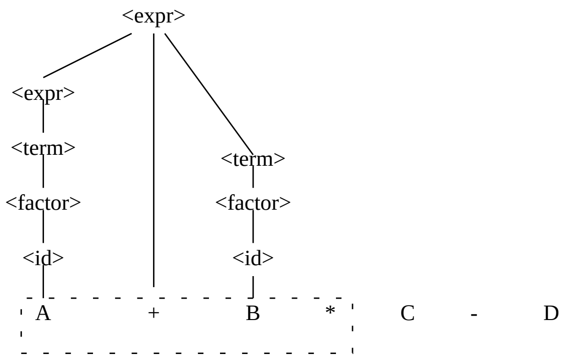
Can't stop -- still have
input tokens left.
Read next token (*)



Parse of A + B * C - D

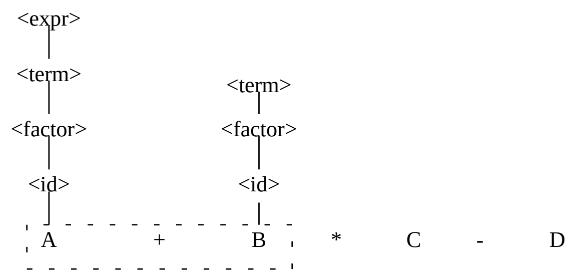
State:
<expr> *
Input left: C - D

No grammar rule produces anything beginning with <expr> * -- dead end. Backtrack to previous state



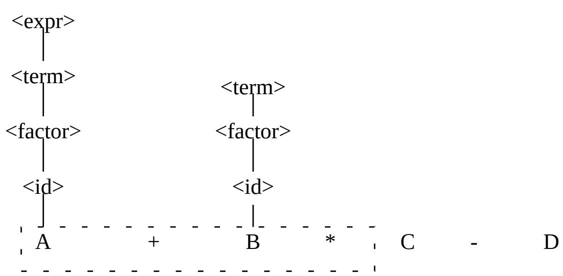
State:
<expr> + <term>
Input left: * C - D

Although we could apply <expr> ::= <expr> + <term>, we've already tried that with no luck -- so read next input token (*).



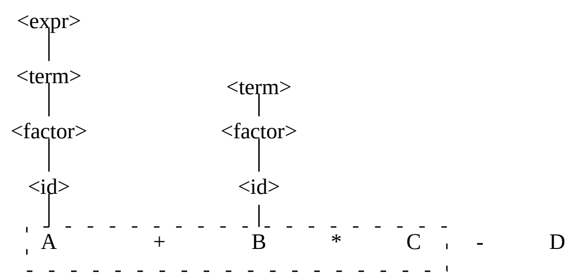
State:
<expr> + <term> *
Input left: C - D

Nothing matches directly with <term> *
Read next token (C).



State:
<expr> + <term> * C
Input left: - D

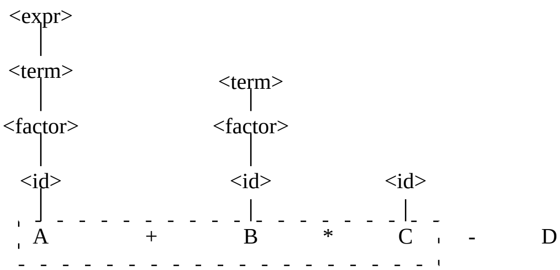
Nothing matches directly with <term> * C
Use grammar rule: <id> ::= C



Parse of A + B * C - D

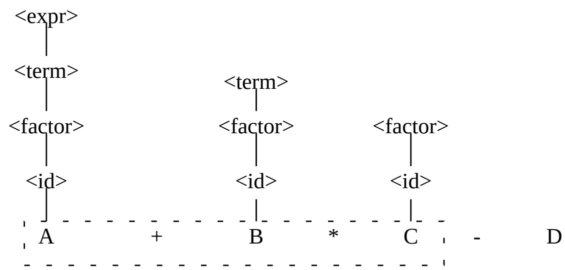
State:
<expr> + <term> * <id>
Input left: - D

Nothing matches directly with <term> * <id>
Use grammar rule: <factor> ::= <id>



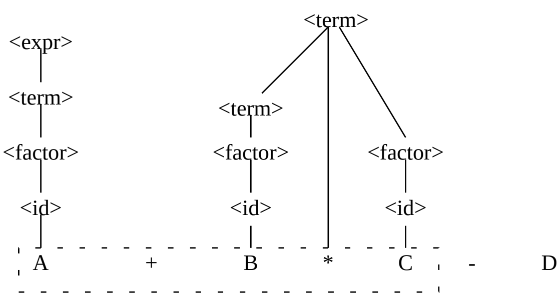
State:
<expr> + <term> * <factor>
Input left: - D

Use grammar rule:
<term> ::= <term> * <factor>



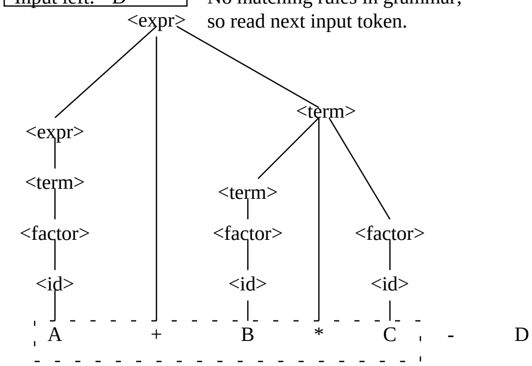
State:
<expr> + <term>
Input left: - D

Use grammar rule:
<expr> ::= <expr> + <term>

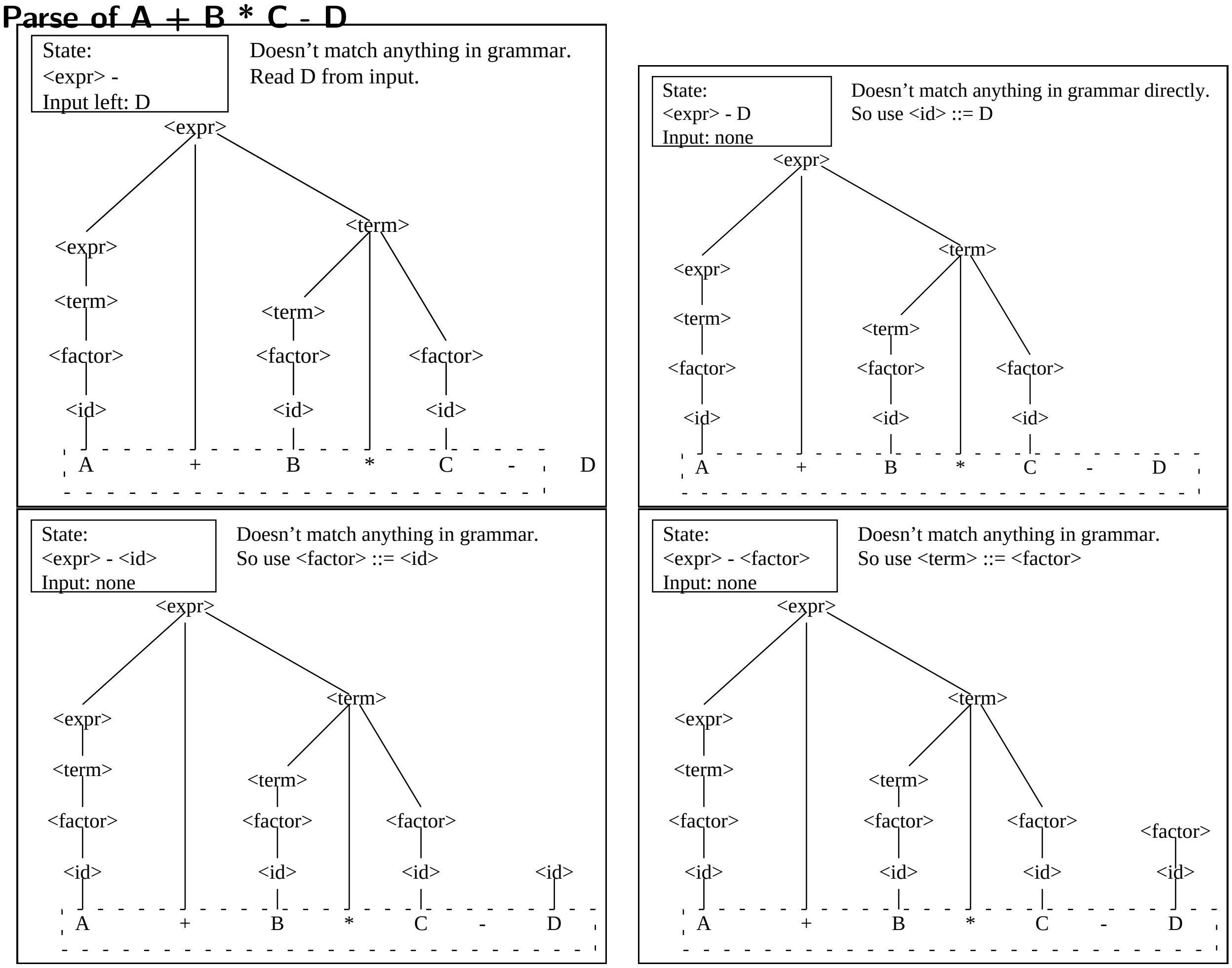


State:
<expr>
Input left: - D

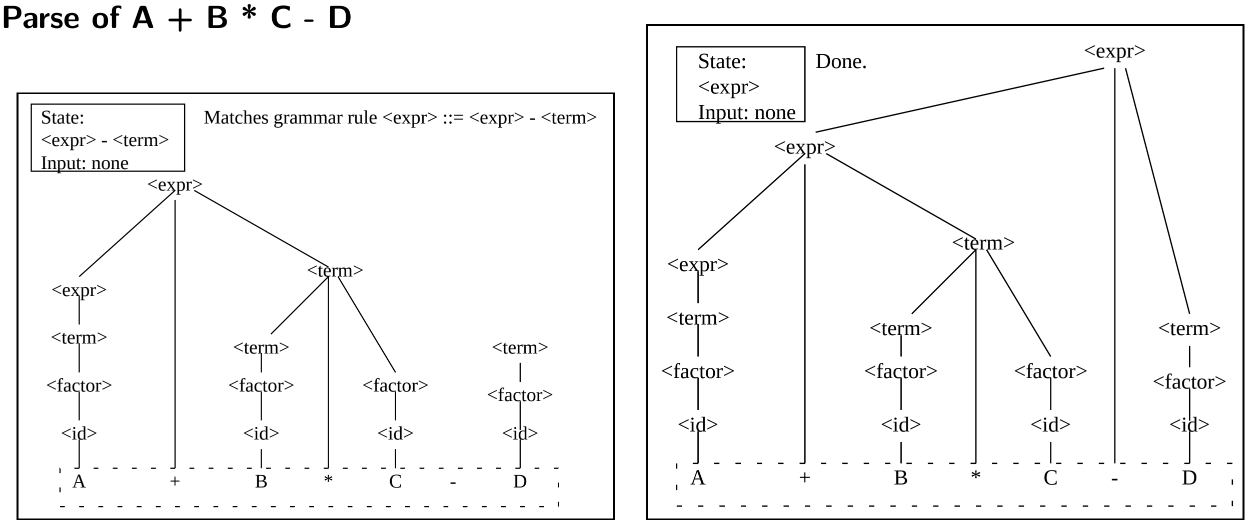
Although <expr> is state, there is still input,
so we’re not done.
No matching rules in grammar;
so read next input token.



Parse of A + B * C - D



Parse of A + B * C - D



Parsers

- Kinds of grammars/kinds of parsing:
  - LL: left-to-right, leftmost derivation
  - LR: left-to-right, rightmost derivation
  - Different amounts of *lookahead*: LR(1), e.g.
- LL parsing: e.g., *recursive-descent parsers*
- LR parsing: e.g., *shift-reduce parsers* – typically table-driven