Grammars and Parsing

CMPUT 275 - Winter 2018

University of Alberta

Parsing

- Parsing is the process of converting a source string into a structured form, known as an Abstract Syntax Tree (AST)

- It consists of two phases:

  - lexical analysis: converts strings of characters into lists of tokens
    (alpha * beta) → ["(", "alpha", "*", "beta", “)"]
    → [LPAR, NAME, BINOP, NAME, RPAR]

    Tokens are divided into classes, such as BINOP (to stand for any binary operator), or NAME (to stand for an variable name)

  - parsing algorithm: takes a grammar (syntax of a language) and a stream of lexical tokens, and produces the AST.

Tokenizing an input string

Tokenizing an input string

- input string:

Tokenizing an input string

- input string:

```
x = 3 + 42 * (s - t)
```

Tokenizing an input string

- input string:

```
x = 3 + 42 * (s - t)
```

- individual tokens:

Tokenizing an input string

- input string:

```
x = 3 + 42 * (s - t)
```

- individual tokens:

```
'x', ‘=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

Tokenizing an input string

- input string:

```
x = 3 + 42 * (s - t)
```

- individual tokens:

```
'x', ‘=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

- pairs of token types and values

Tokenizing an input string

- input string:

```
x = 3 + 42 * (s - t)
```

- individual tokens:

```
'x', ‘=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

- pairs of token types and values

```
('ID','x'), ('EQUALS','='), ('NUMBER','3'), ('PLUS','+'),
('NUMBER','42), ('TIMES','*'), ('LPAREN','('), ('ID','s'),
('MINUS','-'), ('ID','t'), (‘RPAREN',')')
```

Context Free Grammars

- A grammar is collection of recursive rewriting rules (or productions) that say how to combine tokens into grammatical components. The rules are of the form:
  lhs := rhs_0 rhs_1 ... rhs_n

- Productions use two kinds of symbols

  - terminals - which correspond to tokens. Terminals are atomic, in the sense that they are never produced by the grammar, and never appear on a lhs.

  - nonterminals or variables - which correspond to language fragments, or phrases. Variables describe kinds of fragments in the language, and the grammar rules describe how variables are constructed by combining other variables and terminals.

- Both of these are sets. Terminals are sets of tokens. Variables are sets of phrases.

- A grammar has a start symbol, usually a nonterminal.

- A legal string as defined by the grammar is anything that can be generated from the start symbol using productions.

Example

Here is a simple grammar:

```
A := A A
A := a
A := b
```

Example

Here is a simple grammar:

```
A := A A
A := a
A := b
```

```
A = AA                       A = AA
  = aA                         = Ab
  = ab                         = ab
```

Example: Postal code

Here is a simple grammar for postal codes:

```
PCODE := L D L D L D
L := a | b | … | z
D := 0 | 1 | … | 9
```

Example: Postal code

Here is a simple grammar for postal codes:

```
PCODE := L D L D L D
L := a | b | … | z
D := 0 | 1 | … | 9

PCODE := S S S
S := L D
L := a | b | … | z
D := 0 | 1 | … | 9
```

Example: Boolean expressions

Here is a simple grammar for boolean expressions:

```
expr := constant
expr := '(' expr bin_op expr ‘)'
expr := unary_op expr
constant := 'True' | 'False'
bin_op := 'and' | 'or' | 'xor'
unary_op := 'not'
```

Example

Here is a simple grammar for the English language:

```
S → NP VP
VP → V NP
NP → N | D N
N → boy | girl | John | flowers | …
D → a | the
V → sees | likes | …
```

Example

Here is a simple grammar for the English language:

```
S → NP VP
VP → V NP
NP → N | D N | ADJ N | D ADJ N
N → boy | girl | John | flowers | …
ADJ → big | small | …
D → a | the
V → sees | likes | …
```

Example

Here is a simple grammar for the English language:

```
S → NP VP
VP → V NP
NP → N | D N | ADJ N | D ADJ N
N → boy | girl | John | flowers | …
ADJ → big | small | …
D → a | the
V → sees | likes | …
```

Today is a cold day

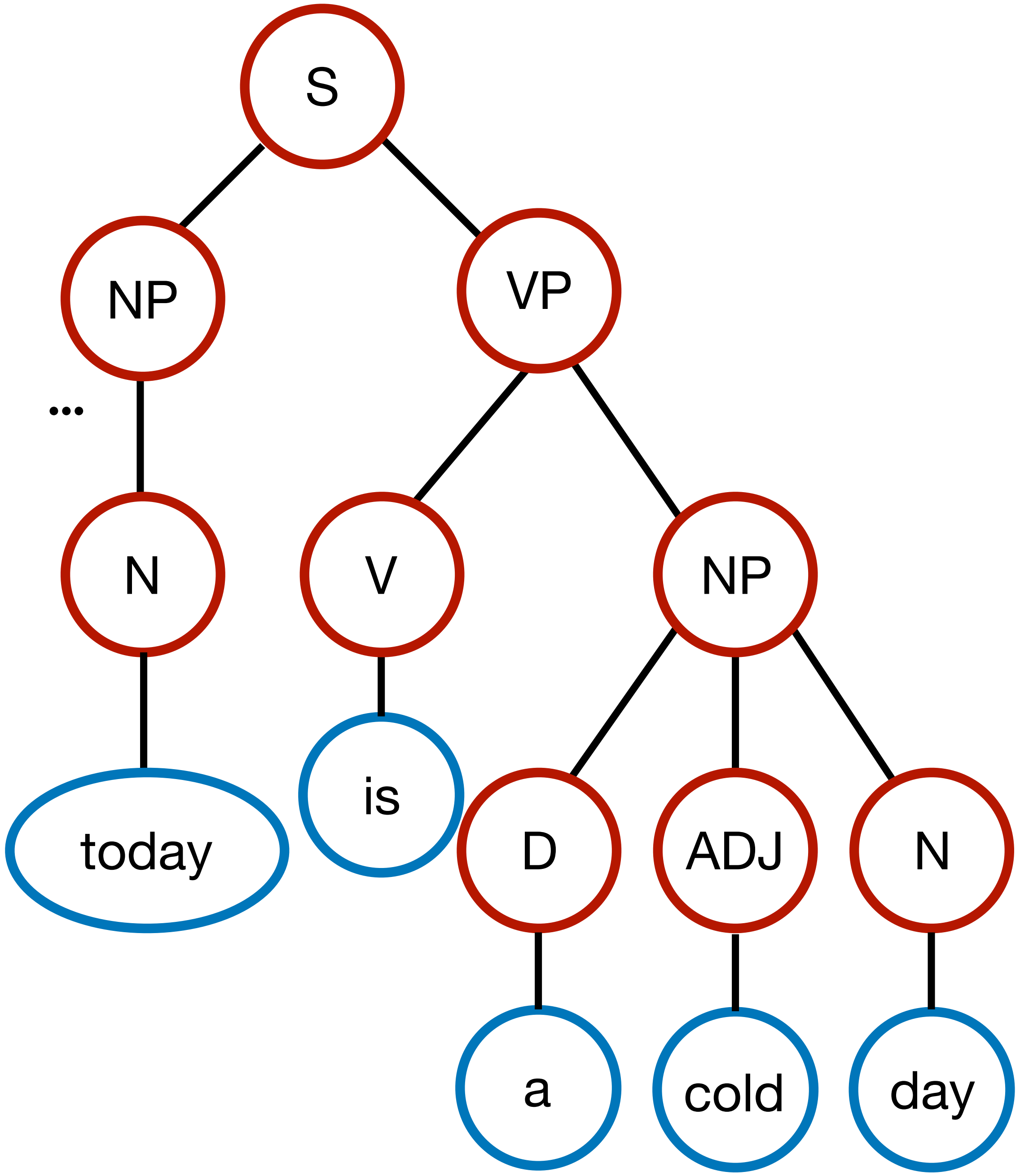
Is it a valid string

(grammatically correct)?

Example

Here is a simple grammar for the English language:

```
S → NP VP
VP → V NP
NP → N | D N | ADJ N | D ADJ N
N → boy | girl | John | flowers | …
ADJ → big | small | …
D → a | the
V → sees | likes | …
```

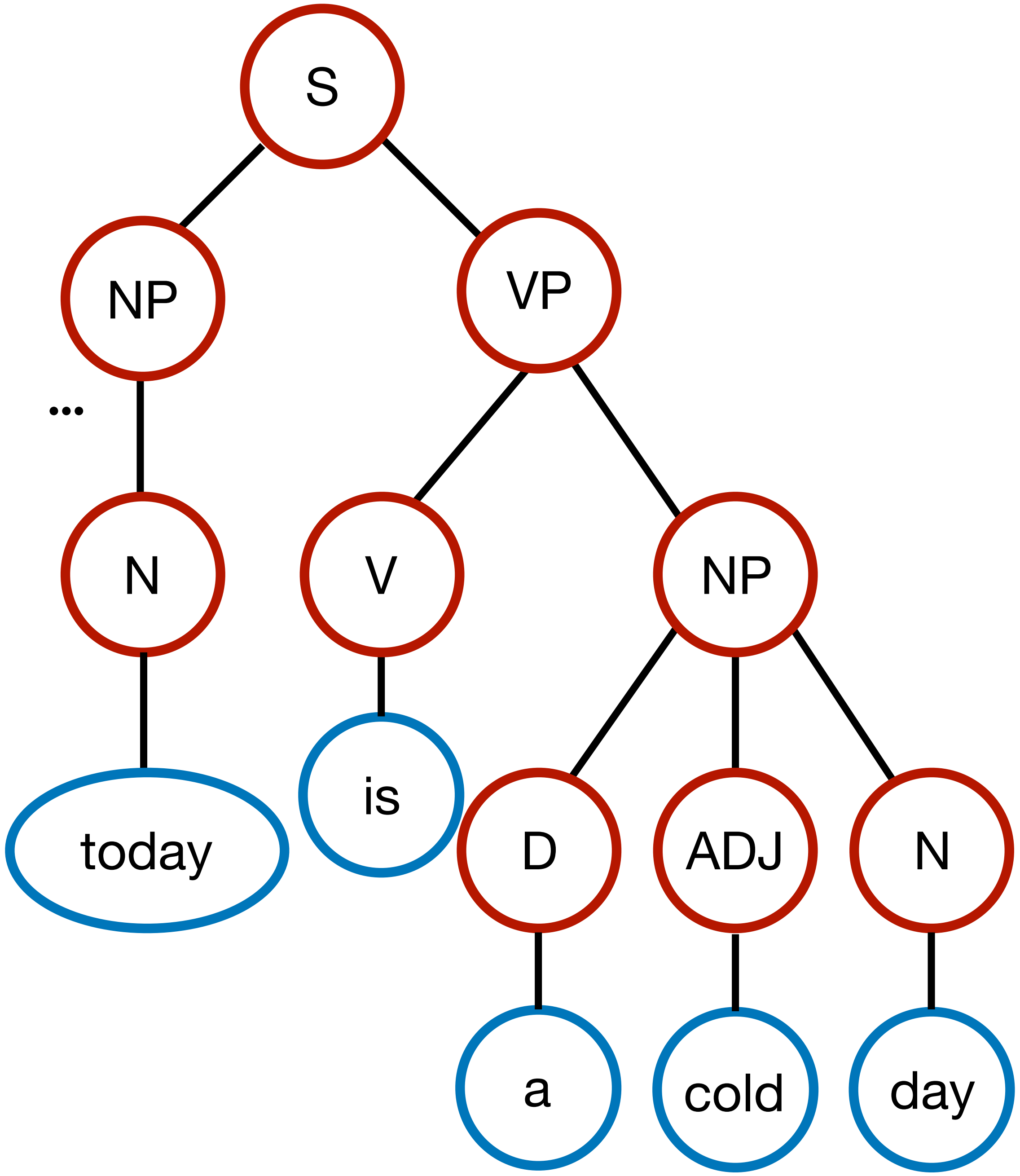


Today is a cold day
Is it a valid string (grammatically correct)?

Example

Here is a simple grammar for the English language:

```
S → NP VP
VP → V NP
NP → N | D N | ADJ N | D ADJ N
N → boy | girl | John | flowers | …
ADJ → big | small | …
D → a | the
V → sees | likes | …
```



Today is a cold sunny day
How about this one?

Example

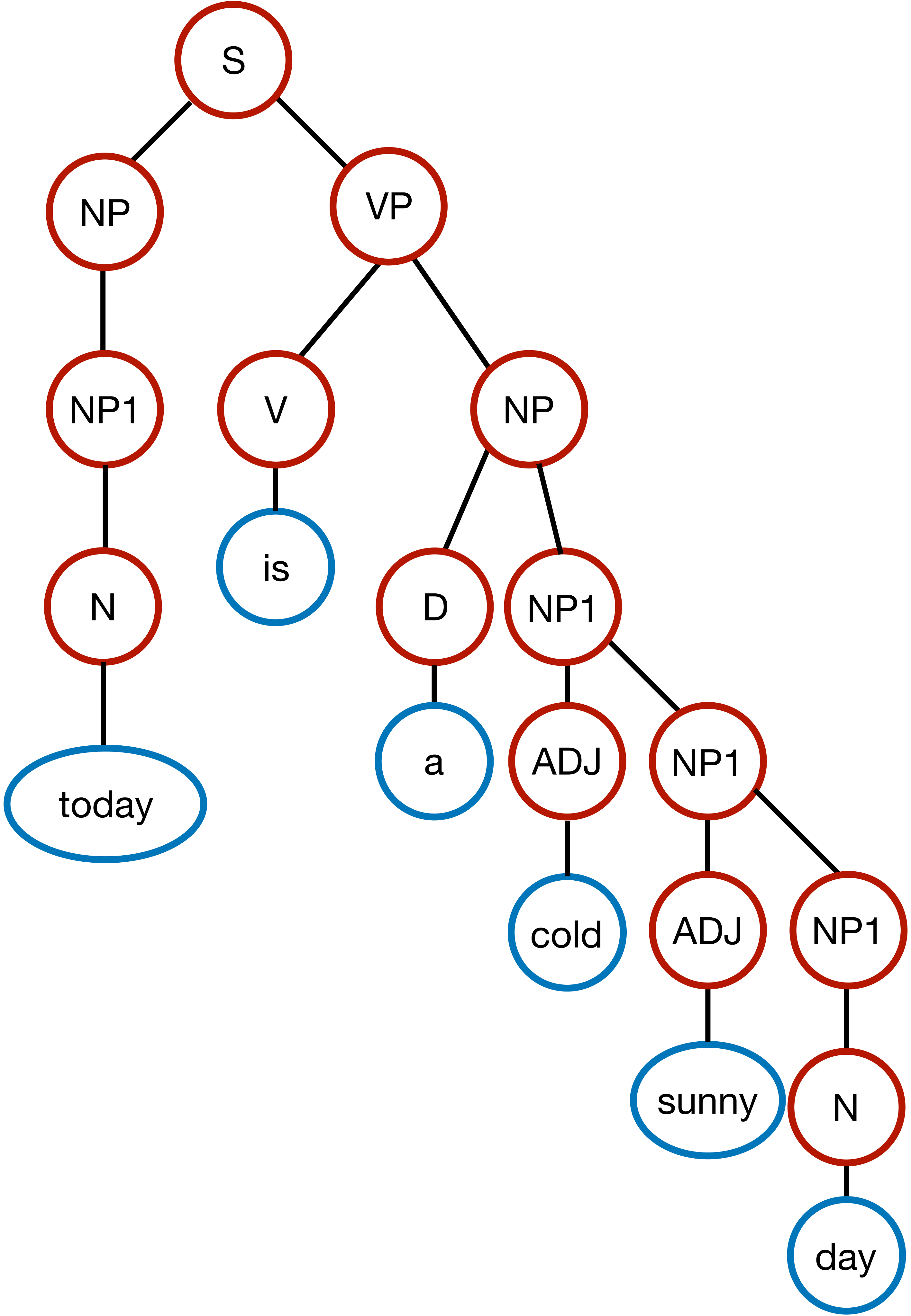
Here is a simple grammar for the English language:

```
S → NP VP
VP → V NP
NP → NP1 | D NP1
NP1 → N | | ADJ NP1
N → boy | girl | John | flowers | …
V → sees | likes | …
ADJ → big | small | …
D → a | the
```

Example

Today is a cold sunny day

How about this one?



Example

Tokens

```
literals = ['=', '+', '-', '*', '/', ]
NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
COMMA = r','
LPAREN = r'\('
RPAREN = r'\)'
NUMBER = r'\d+'
```

Grammar

```
S' -> statement
statement -> expression
statement -> <empty>
expression -> NAME = expression
expression -> expression + expression
expression -> expression - expression
expression -> expression * expression
expression -> expression / expression
expression -> - expression
expression -> NAME LPAREN RPAREN
expression -> NAME LPAREN expr_list RPAREN
expression -> LPAREN expression RPAREN
expr_list -> expression
expr_list -> expr_list COMMA expression
expression -> NUMBER
expression -> NAME
```

Parser Generator

- Defining the grammar for a complex language is a difficult and error-prone process. Hand crafting a parser for a grammar is even worse.

- Thus tools for converting grammars into parsers have been developed, commonly called compiler-compilers (then compile a description into a compiler). ***yacc*** stands for yet-another-compiler-compiler. The actual mechanics of parsing is quite simple to describe in terms of the operation of a type of machine called a push-down automata (or stack machine). It's getting the specific rules right that is tricky.