Pythonic Parsing with Pyparsing

Dr Andrew Lawrence

PyCon UK 2018

Cardiff, 17 September 2018

Who Am I?

- Software Engineer

Who Am I?

- Software Engineer
- Previously, I did a PhD looking at automatic formal verification of train control systems

Who Am I?

- Software Engineer
- Previously, I did a PhD looking at automatic formal verification of train control systems
- First PyCon

Who Am I?

- Software Engineer
- Previously, I did a PhD looking at automatic formal verification of train control systems
- First PyCon
- First workshop

Workshop Overview

This workshop aims to give an introduction to parsing using the Pyparsing library

Overview:

- Part I: Parsing Introduction
- Part II: Pyparsing Basics
- Part III: Exercise 1: Dice Rolling
- Part IV: Intermediate Pyparsing
- Part V: Exercise 2: JSON Parser
- Part VI: Advanced Pyparsing

Prerequisites

I am assuming that you have **Python 3** and **git** installed. You can get the workshop examples from Github:

git clone https://github.com/andrewjlawrence/pyparsingworkshop.git

You also need to install Pyparsing.

pip install pyparsing

Why do we need parsers?

The world is full of textual data in structured human readable formats.

- XML
- JSON
- other propriety formats...

The problem here is getting computers to understand this data.

Why do we need parsers?

```
<customer id="cust0921">
  <first-name>Andrew</first-name>
  <last-name>Lawrence</last-name>
  <address>
    <street>Langley Park</street>
    <city>Chippenham</city>
    <county>Wiltshire</county>
    <postcode>SN122BL</postcode>
  </address>
</customer>
```

What is a parser?

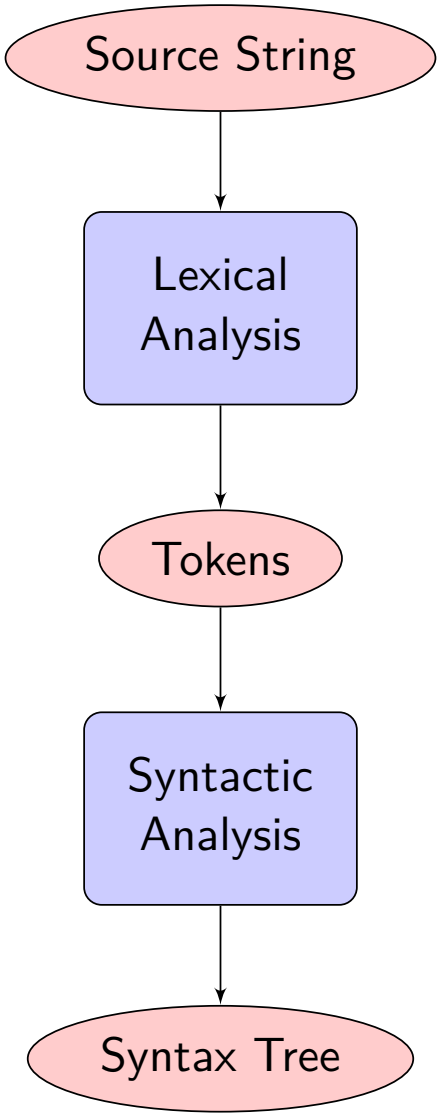
Typically parsing involves 3 stages:

Lexical Analysis Break the string down into **tokens**

Syntactic Analysis Construct a **syntax tree** based on some grammar

Semantic Interpretation Do something based on the syntax tree

What is a parser?



What is a Grammar?

A **grammar** is a set of rules that describes the structure of text.

It specifies the syntax that should be accepted by the parser.

Backus-Naur Form (BNF)

A Backus-Naur form (BNF) is metasyntax notation for describing grammars.

element type	description
< nonterminals >	intermediate labels
terminals	printable characters
|	choice
::=	replaced-by

Arithmetic Example Grammar

$$\langle sum \rangle ::= \langle sum \rangle + \langle product \rangle \mid \langle product \rangle$$

$$\langle product \rangle ::= \langle product \rangle * \langle value \rangle \mid \langle value \rangle$$

$$\langle value \rangle ::= \langle int \rangle \mid id$$

$$\langle int \rangle ::= \langle unsignedint \rangle \mid -\langle unsignedint \rangle$$

$$\langle unsignedint \rangle ::= \langle digit \rangle \mid \langle unsignedint \rangle \langle digit \rangle$$

$$\langle digit \rangle ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Regular Expressions

A *regular expression* is a syntactic notation for defining textual patterns.

Example: `a|b*` denotes $\{\epsilon, "a", "b", "bb", "bbb", \ldots\}$

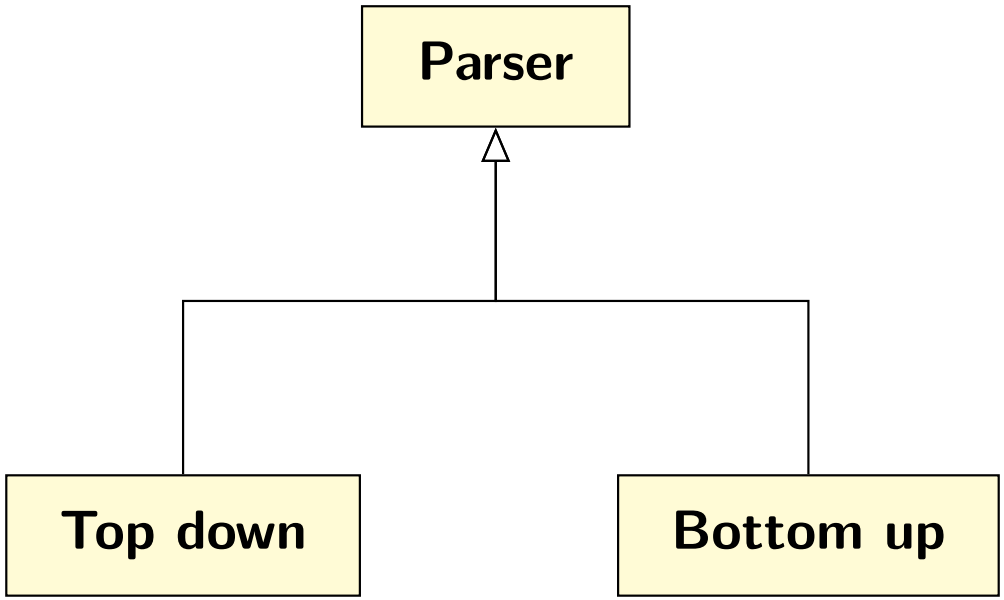
I won't go into further detail about how these work but beaware that they exist.

BNF Extensions

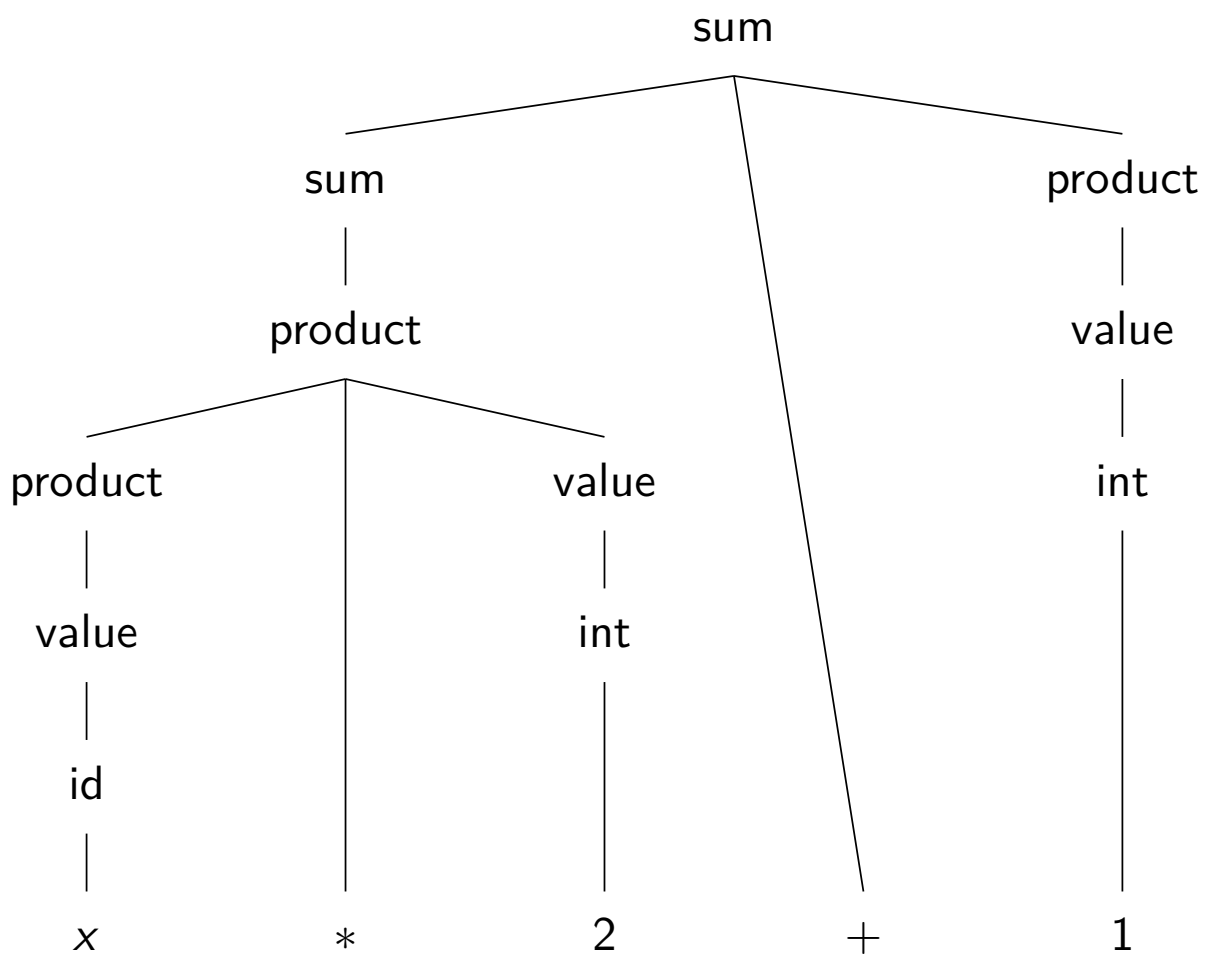
There are a number of extensions to BNF. One common approach, as seen in extended Backus-Naur form is to use operations from regular expressions such as:

- `*` - Match the preceeding element zero or more times. Often called the **Kleene Star**.
- `+` - Match the preceeding element one or more times

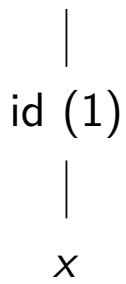
Types of Parsers



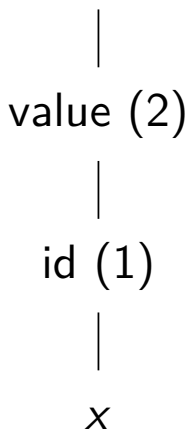
Example Parse Tree



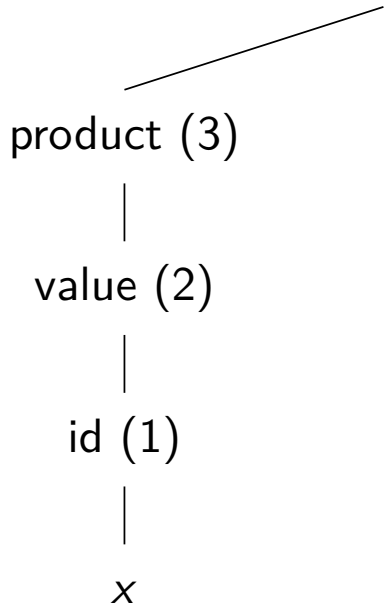
Bottom Up (LR) Parse



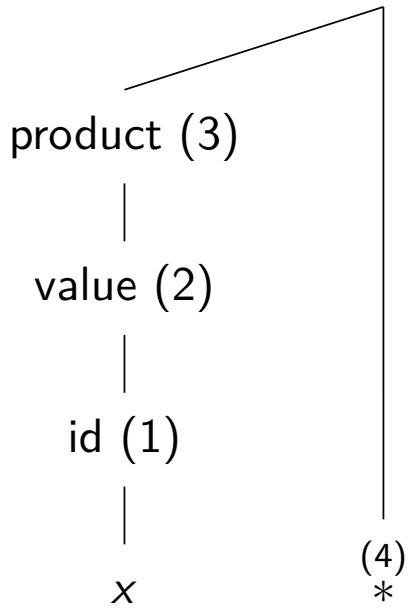
Bottom Up (LR) Parse



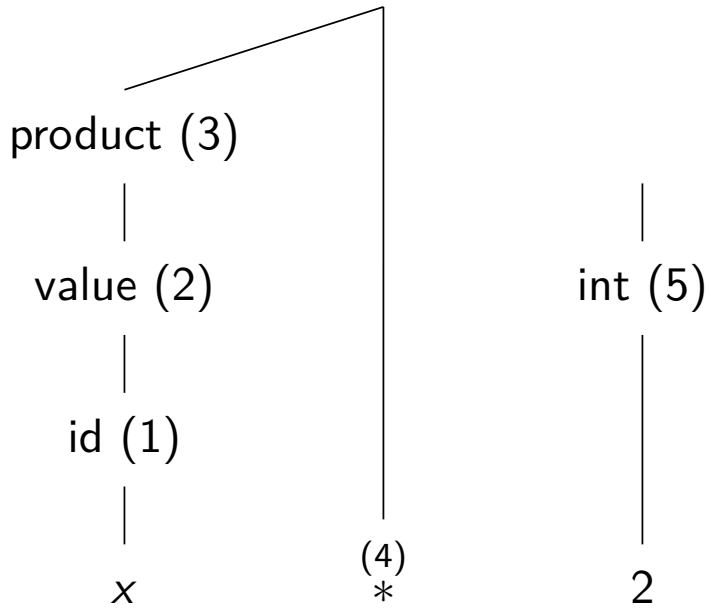
Bottom Up (LR) Parse



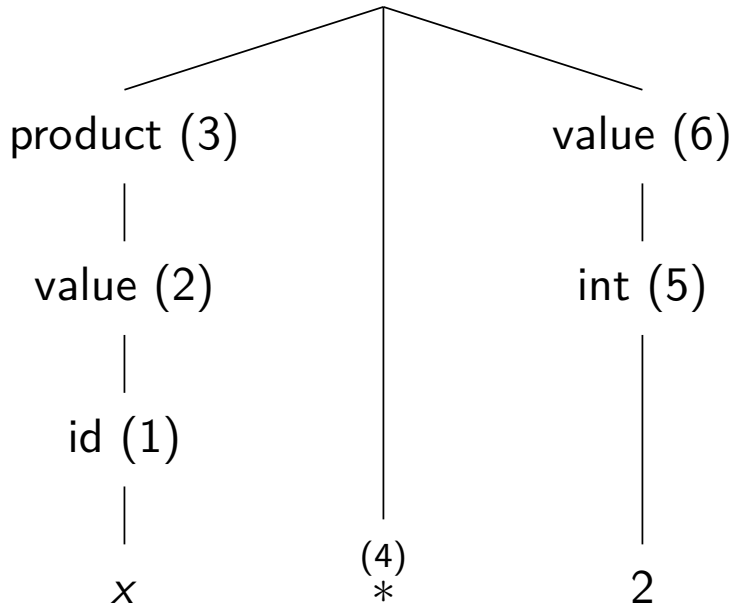
Bottom Up (LR) Parse



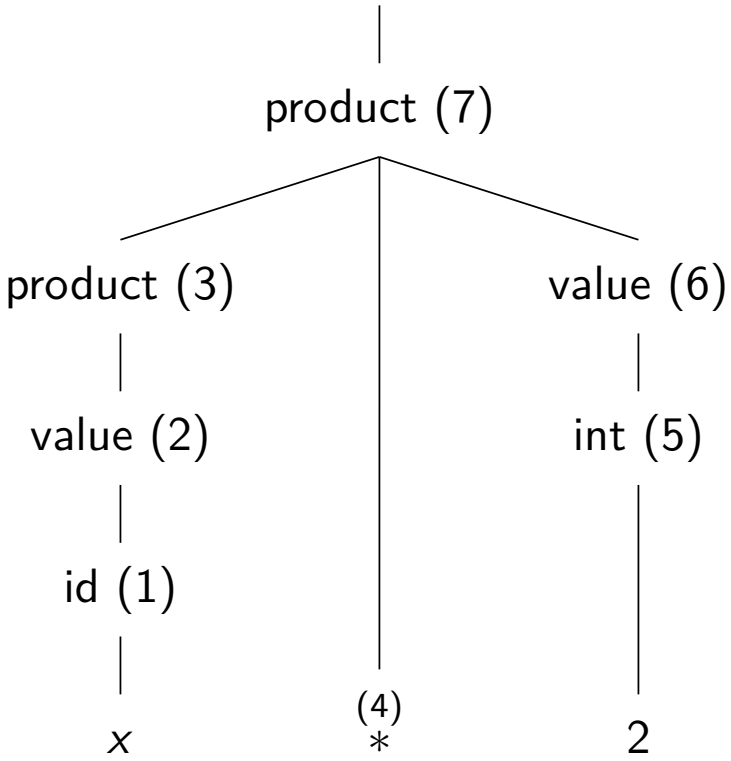
Bottom Up (LR) Parse



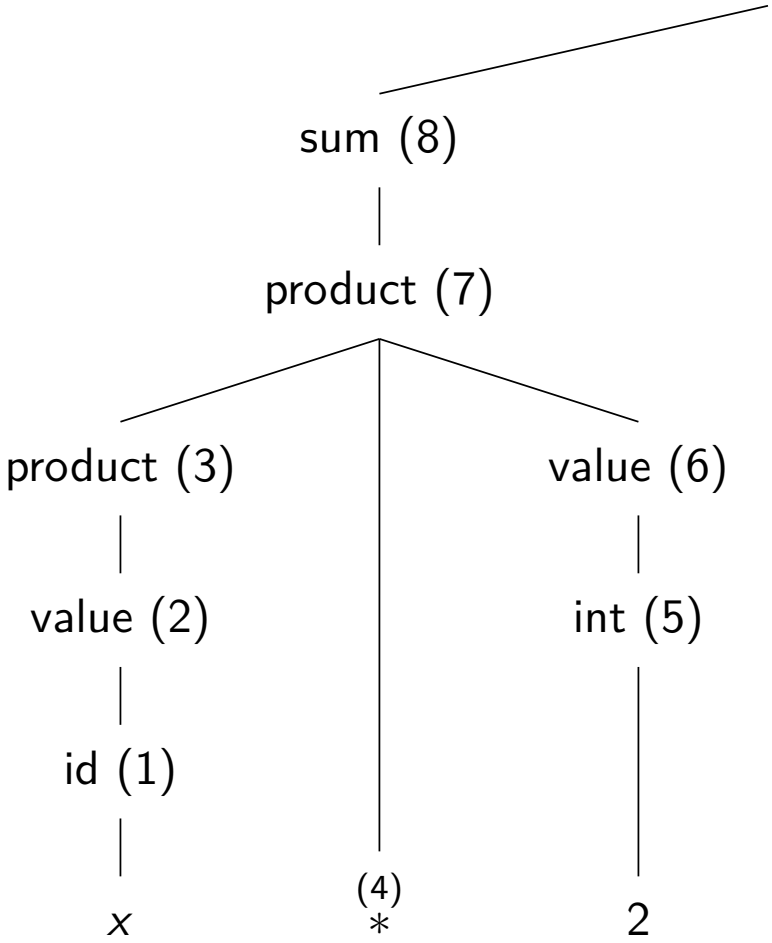
Bottom Up (LR) Parse



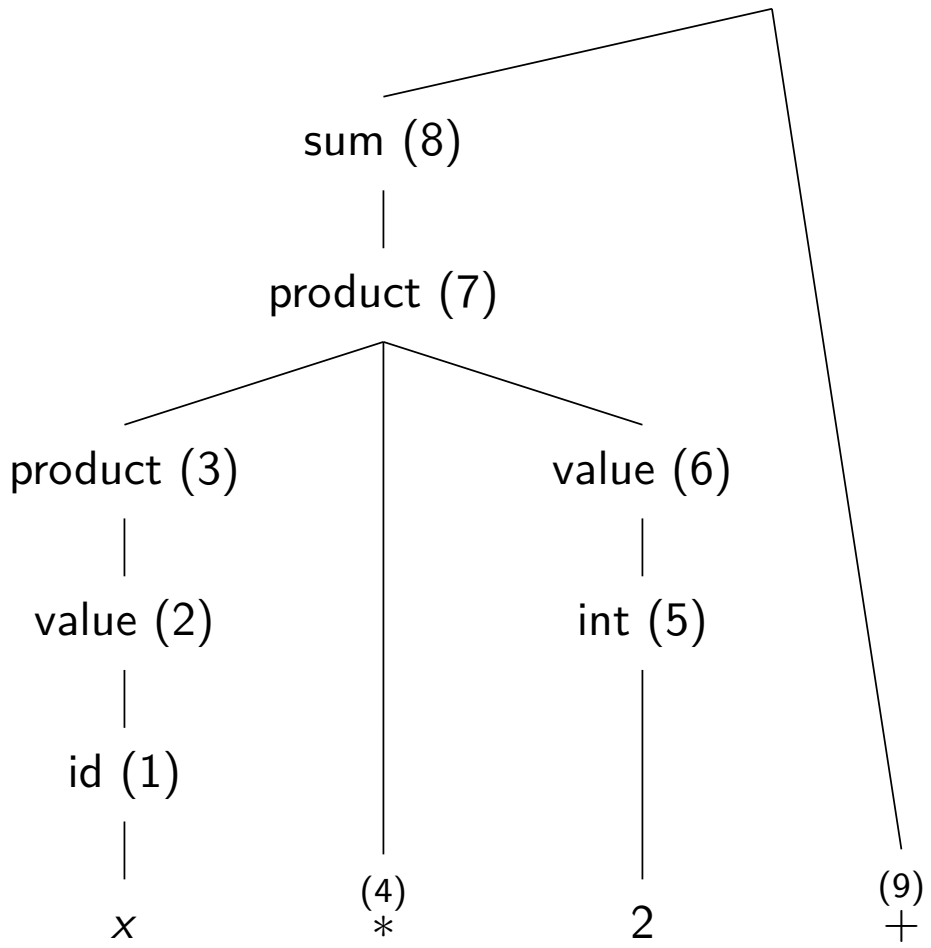
Bottom Up (LR) Parse



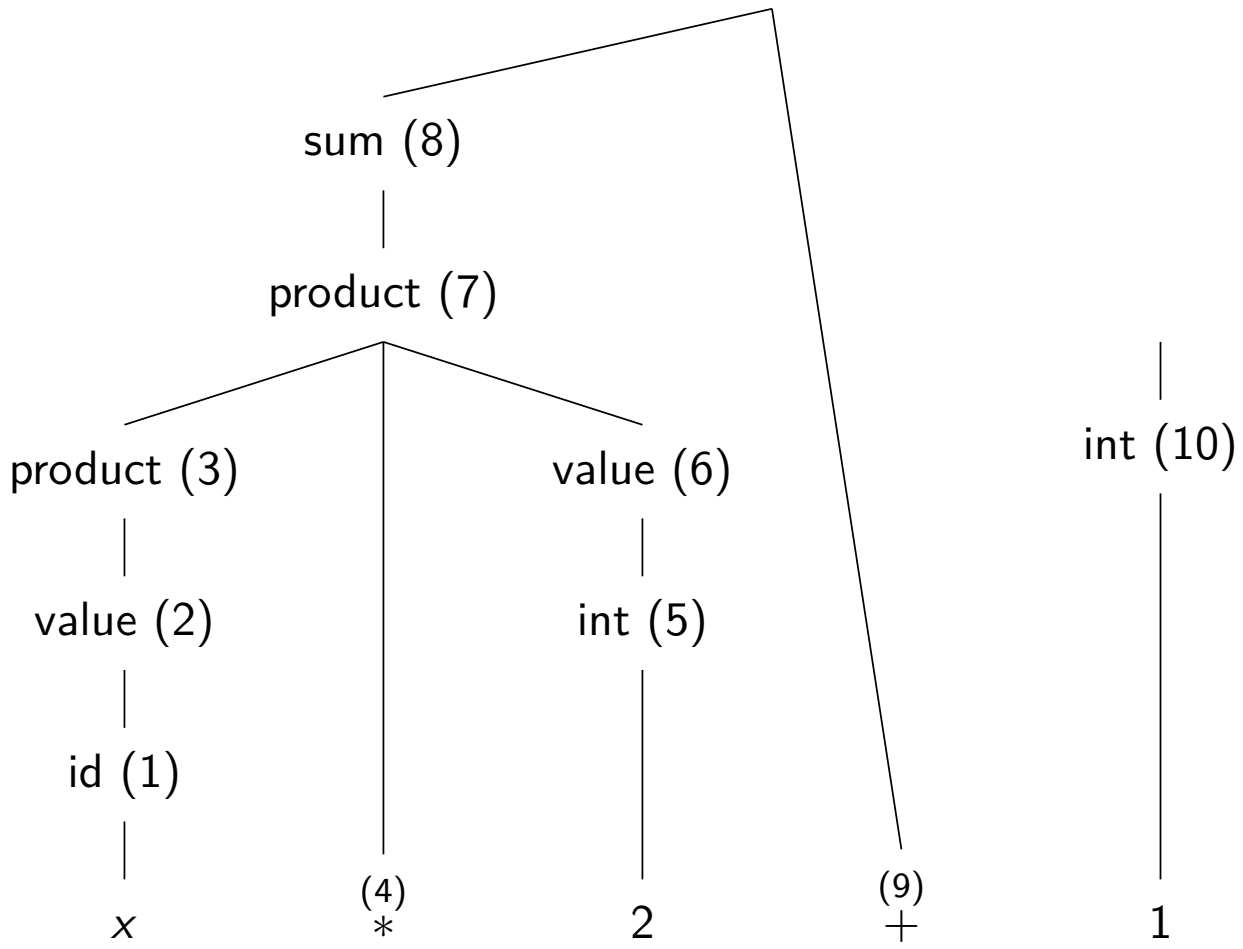
Bottom Up (LR) Parse



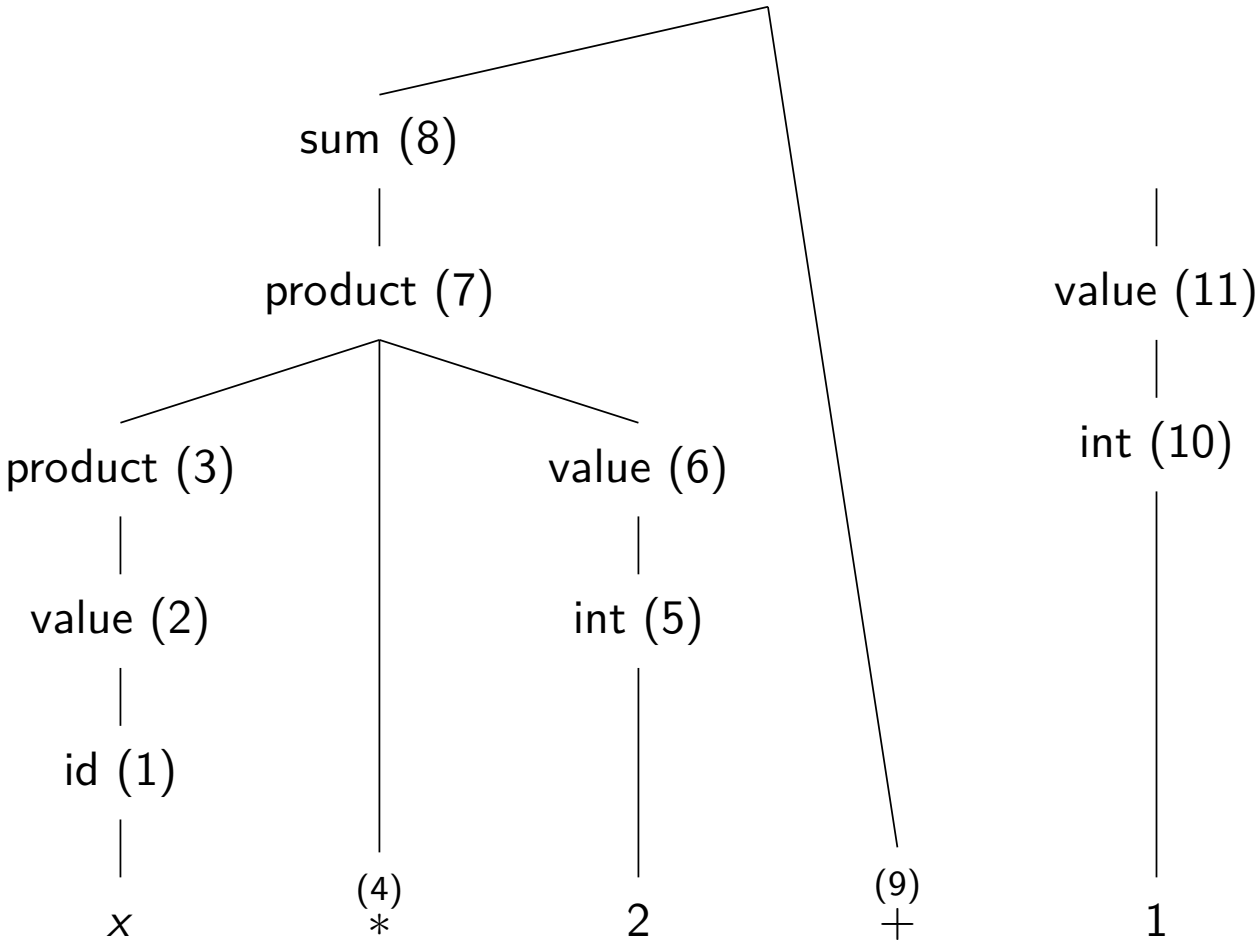
Bottom Up (LR) Parse



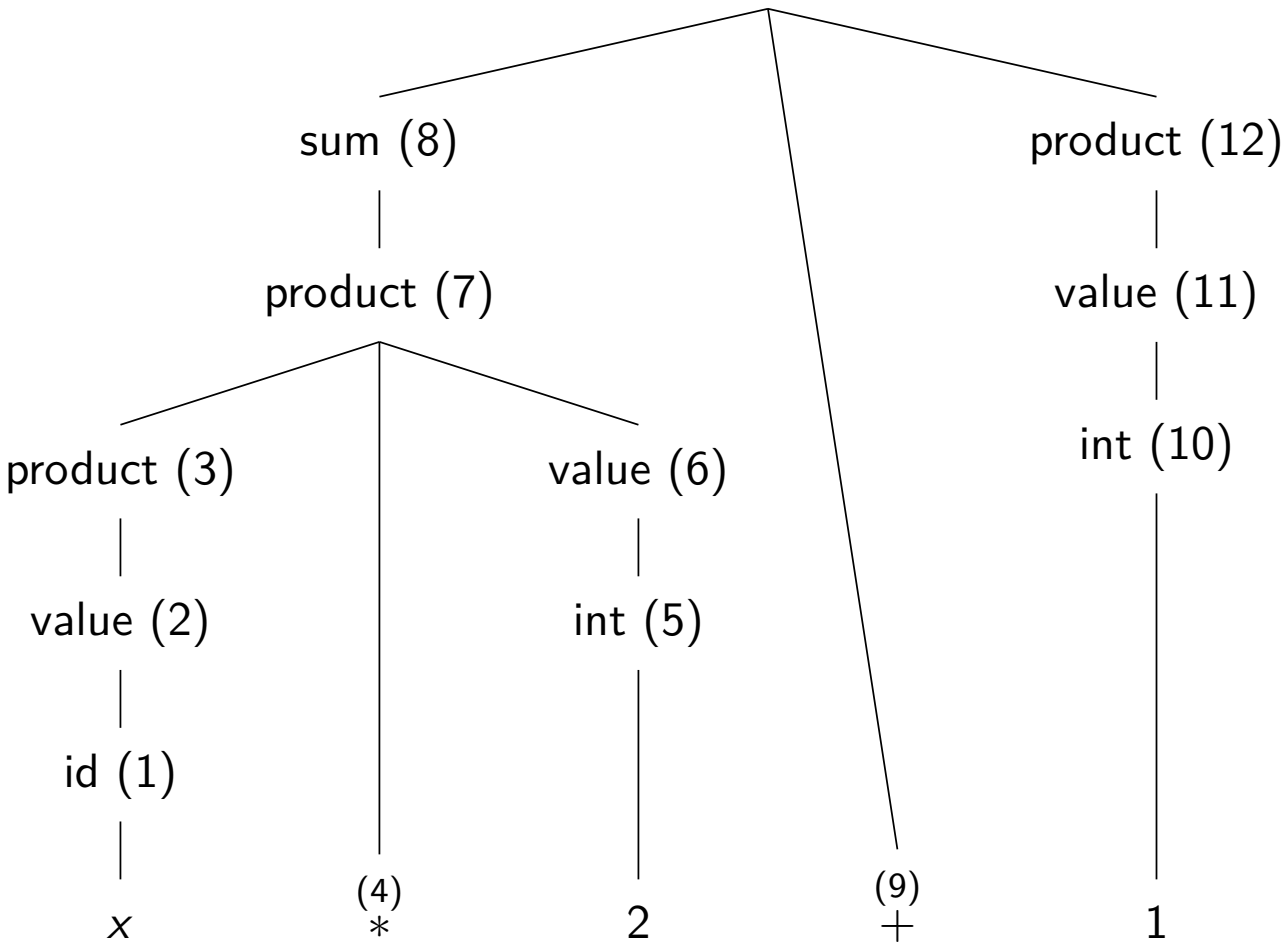
Bottom Up (LR) Parse



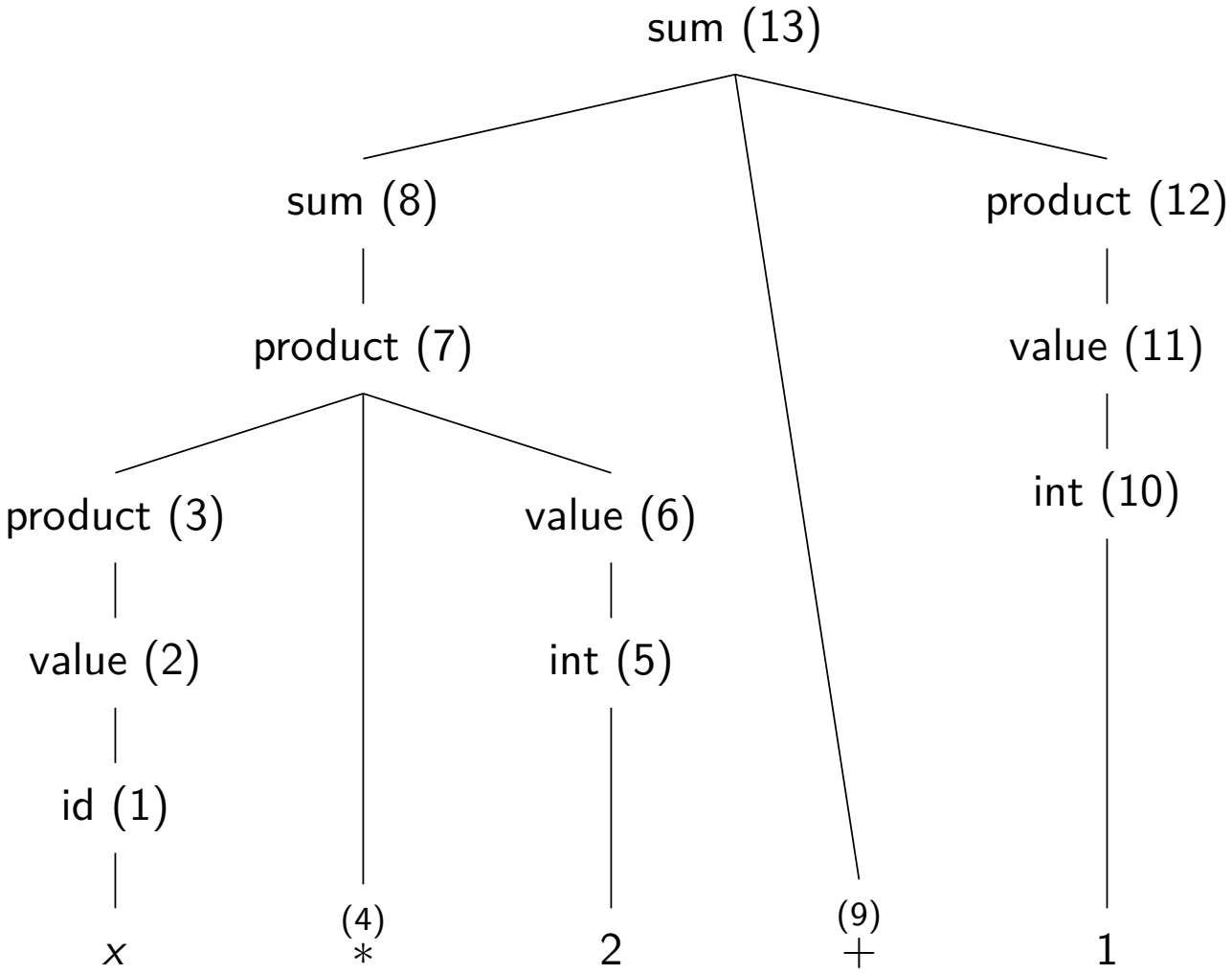
Bottom Up (LR) Parse



Bottom Up (LR) Parse



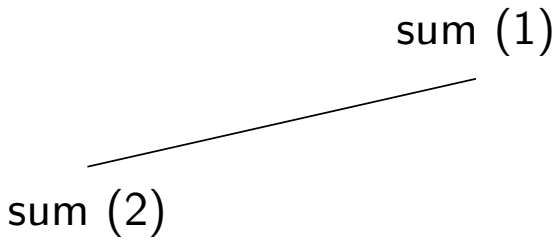
Bottom Up (LR) Parse



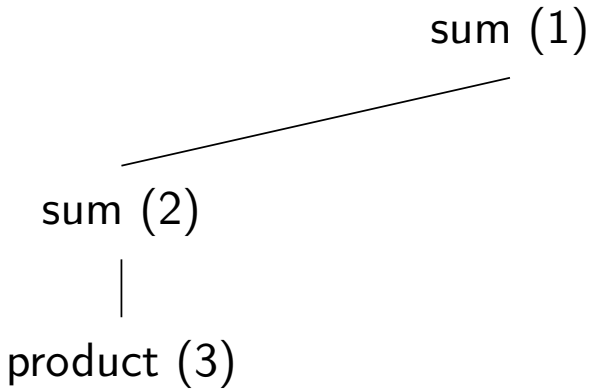
Top Down Parse

sum (1)

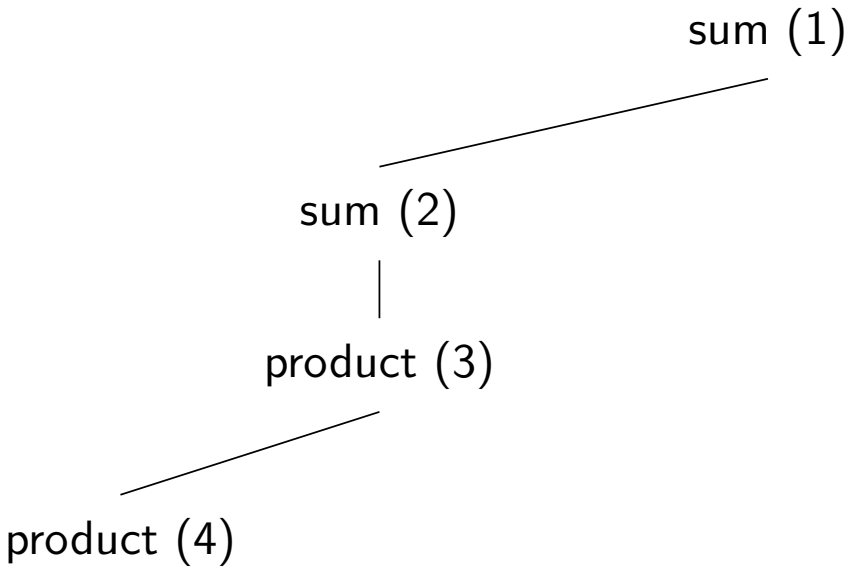
Top Down Parse



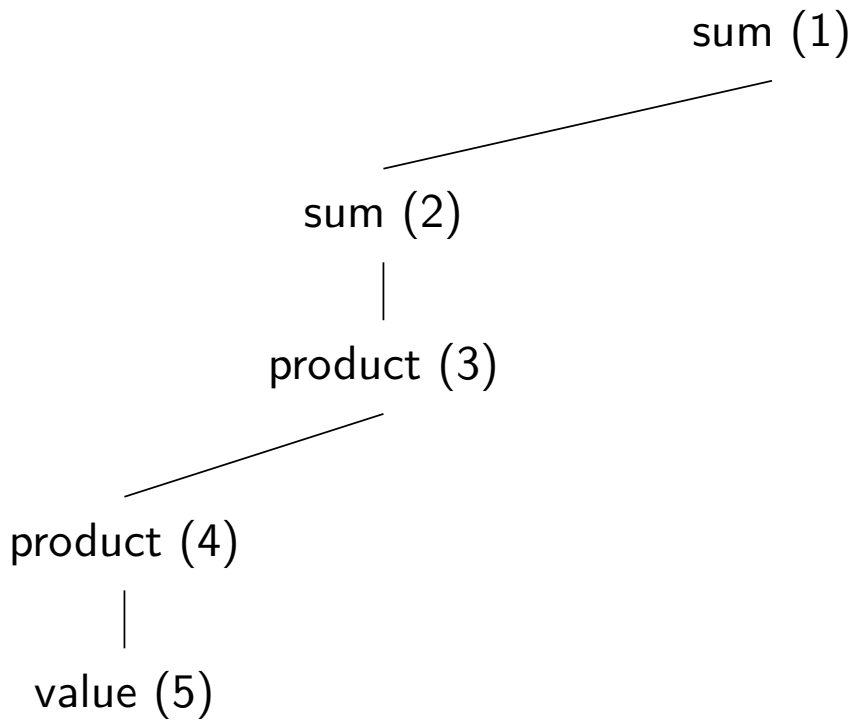
Top Down Parse



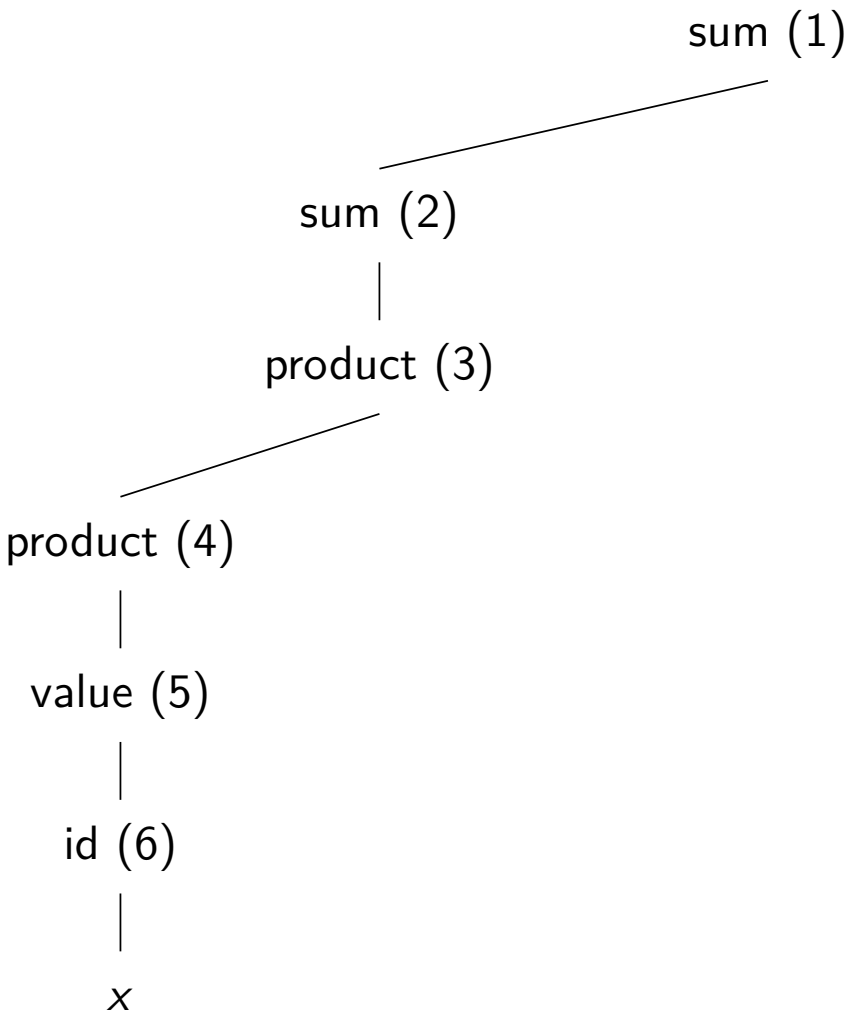
Top Down Parse



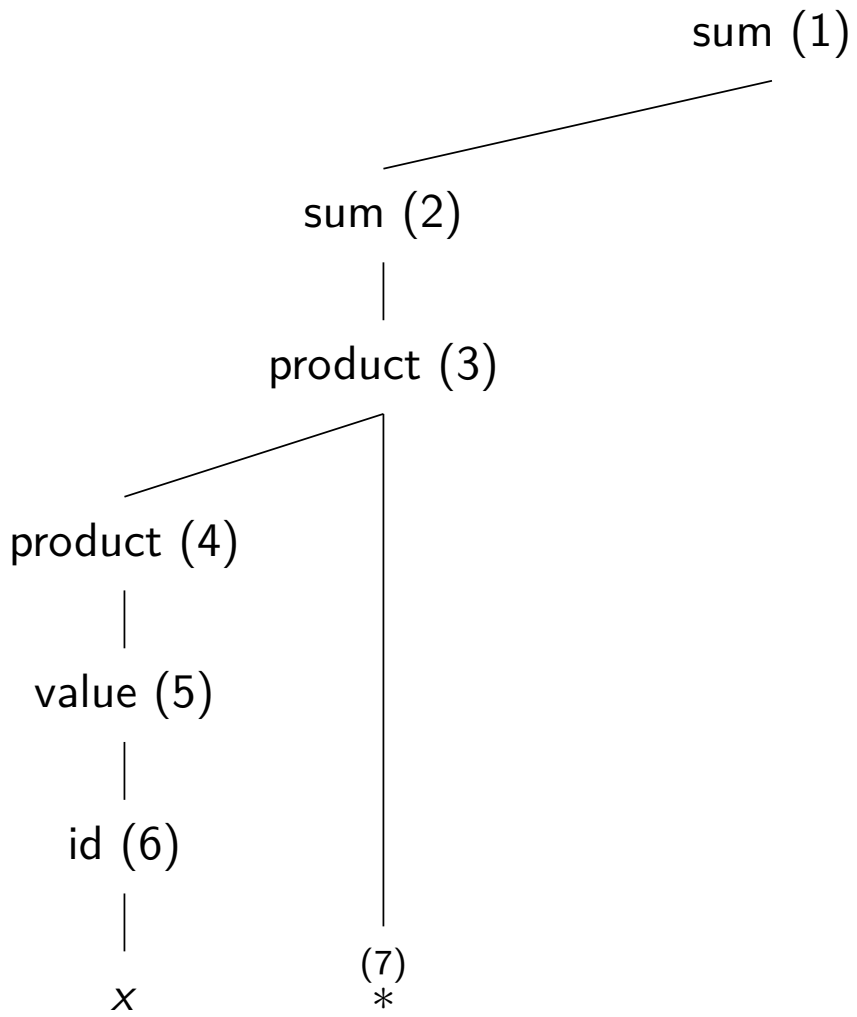
Top Down Parse



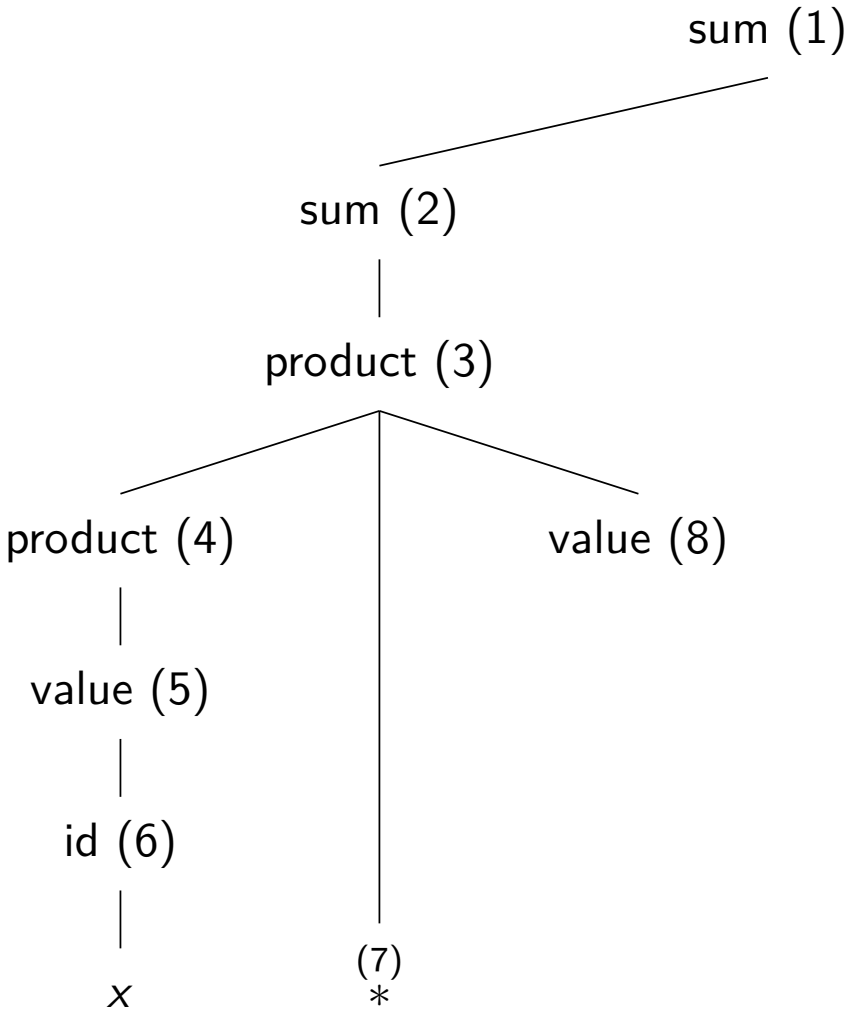
Top Down Parse



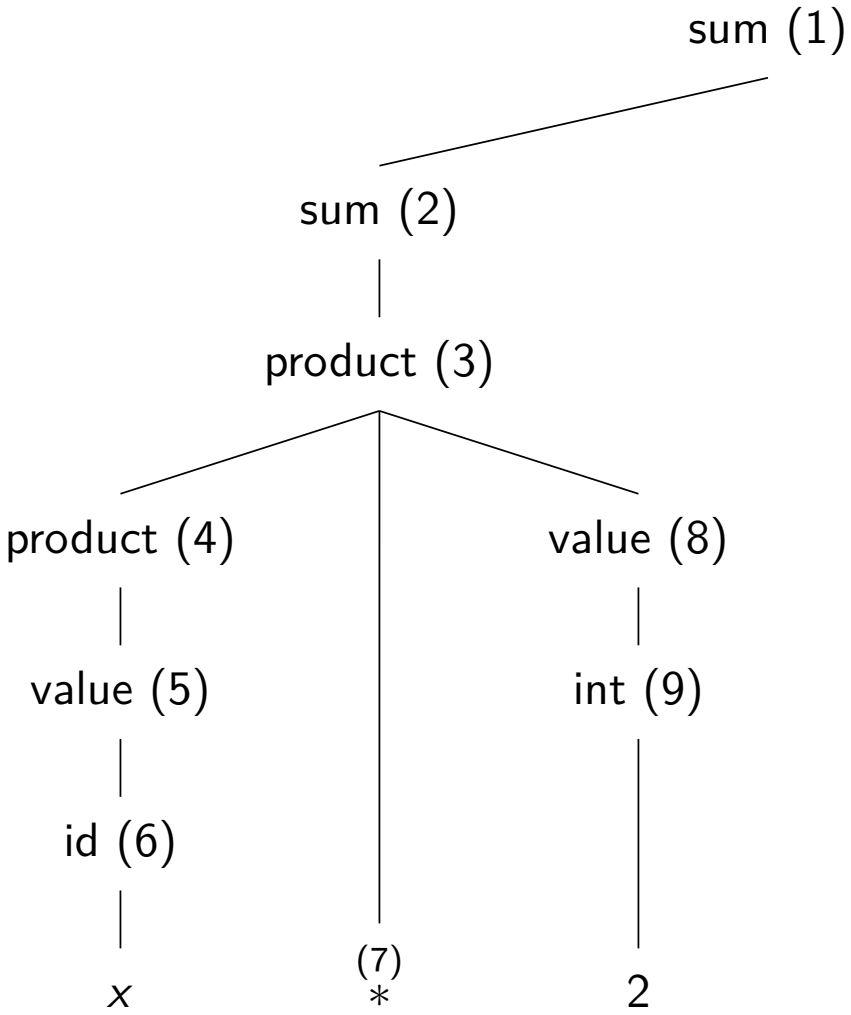
Top Down Parse



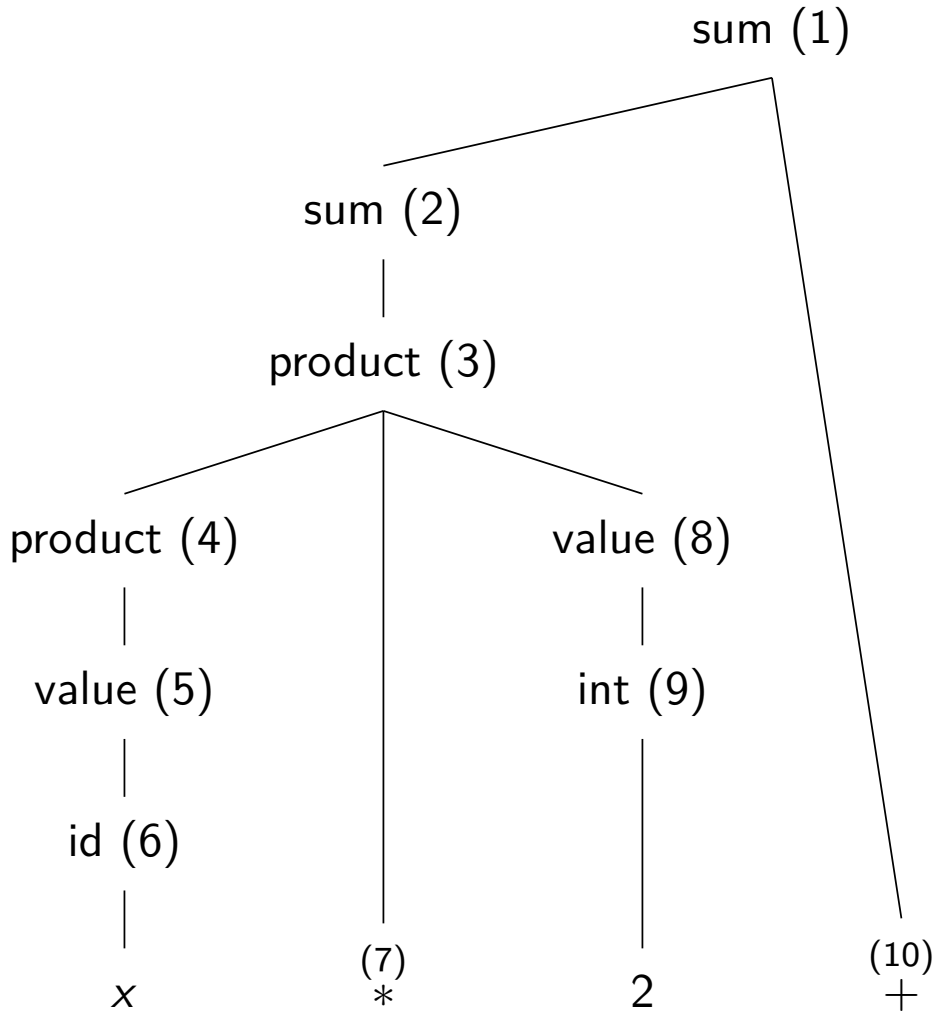
Top Down Parse



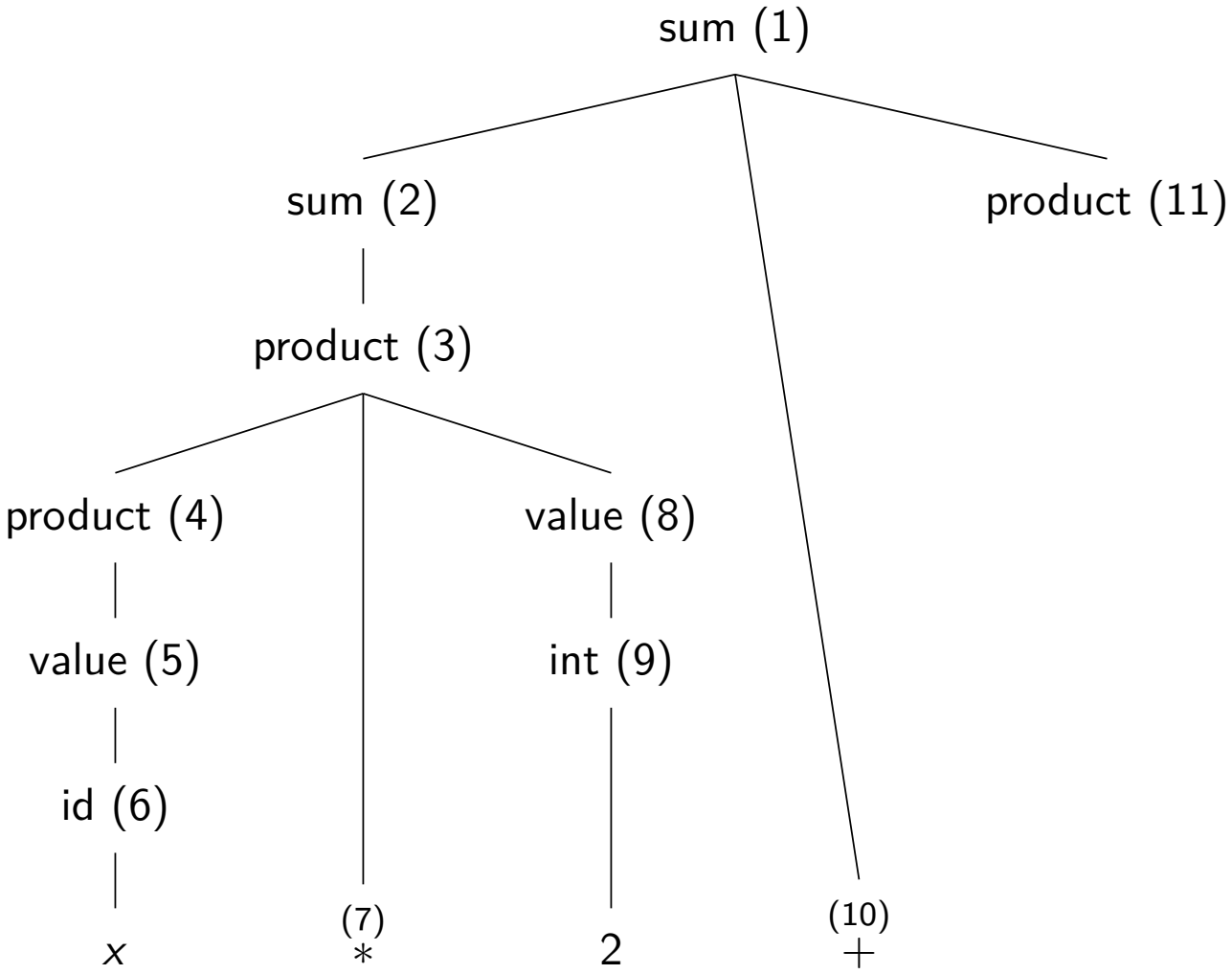
Top Down Parse



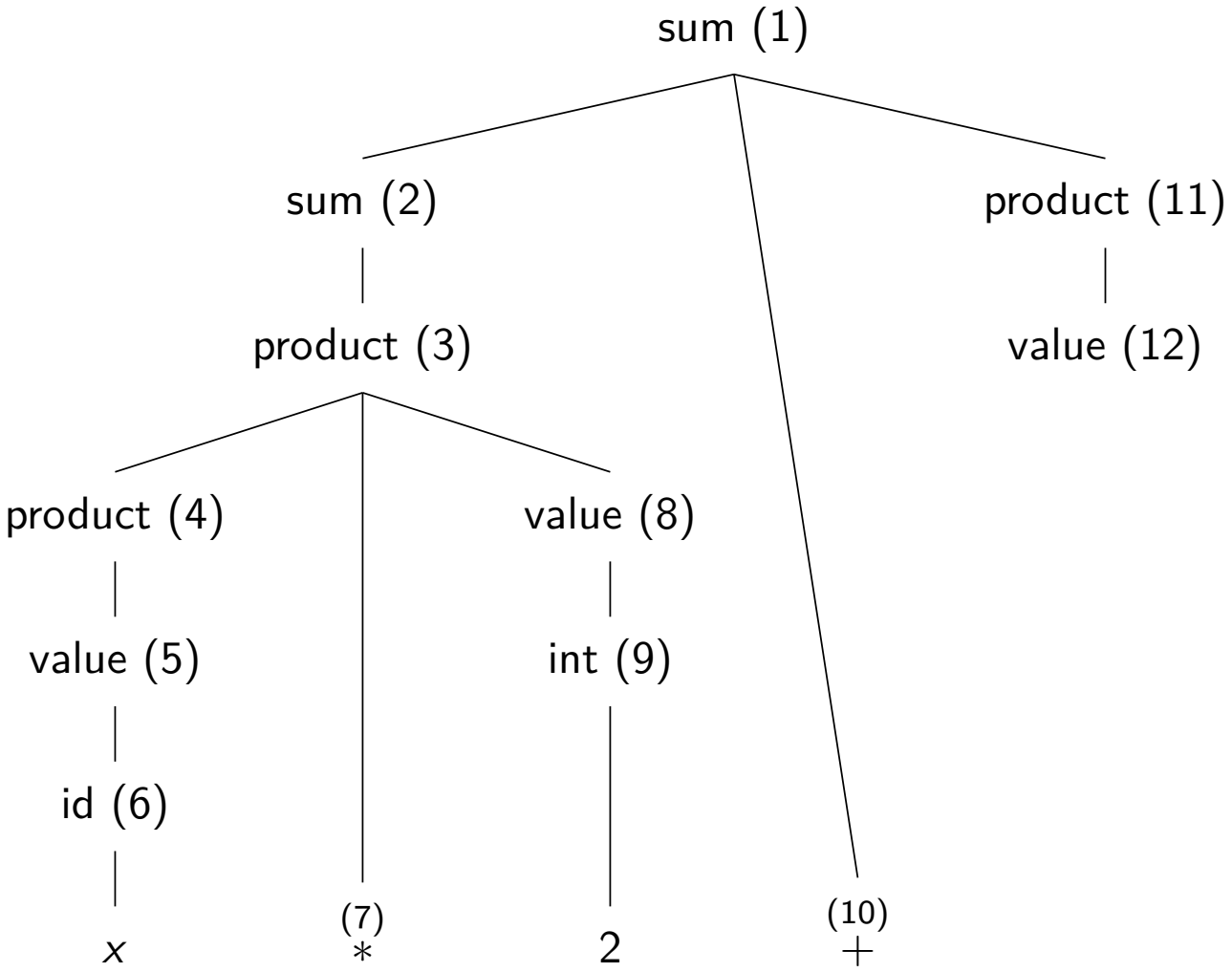
Top Down Parse



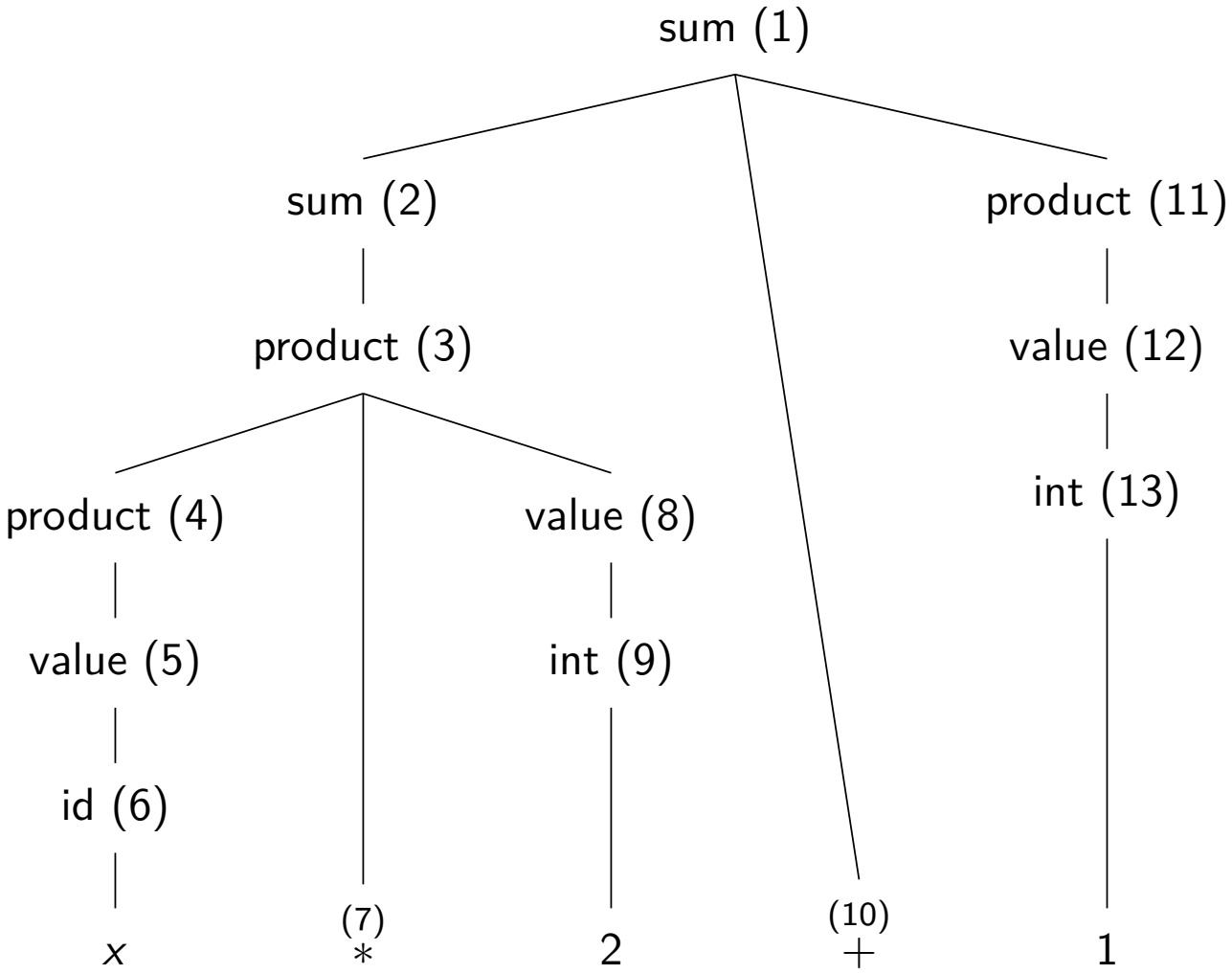
Top Down Parse



Top Down Parse



Top Down Parse



Why Are Regular Expressions Bad You Ask?

Below is part of a regex for validating RFC822 email addresses.

```
(?:(?:\\r\\n)?[ \\t])*(?:(?:(?:[^()<>@,;:\\\\".\\[\\]
 \\000-\\031]+(?:(?:(?:\\r\\n)?[ \\t])+|\\Z|
(?=[\\["()<>@,;:\\\\".\\[\\]]))|"(?:[^\\\\"\r\\\\]|\\\\.|
(?:(?:\\r\\n)?[ \\t]))*"(?:(?:\\r\\n)?[ \\t])*)
(?:\\.(?:(?:\\r\\n)?[ \\t])
*(?:[^()<>@,;:\\\\".\\[\\] \\000-\\031]+
(?:(?:(?:\\r\\n)?[ \\t])+|\\Z|
(?=[\\["()<>@,;:\\\\".\\[\\]]))|"
(?:[^\\"\r\\\\]|\\\\.|(?:(?:\\r\\n)?[ \t]))
*"(?:(?:\\r\\n)?[ \\t])*))*@(?:(?:\\r\\n)?[ \\t])*
```

The full regex is 6.2KB in size.

Why Are Regular Expressions Bad You Ask?

Why are regular expressions bad?

Why Are Regular Expressions Bad You Ask?

Why are regular expressions bad?

1. Difficult to read

Why Are Regular Expressions Bad You Ask?

Why are regular expressions bad?

1. Difficult to read
2. Hard to maintain - **write only**

Why Are Regular Expressions Bad You Ask?

Why are regular expressions bad?

1. Difficult to read
2. Hard to maintain - **write only**
3. Different standards exist

Pyparsing Overview

Pyparsing is a recursive decent parser framework for the Python programming language.

Created by Paul McGuire in 2003.

Follows the parser combinator approach to parsing.

Parser Combinator Approach

You build your parser by combining other parsers.

$$Parser_A = Parser_B + Parser_C$$

Pyparsing Advantages

Comprehensive framework - Contains many predefined parsers.

Powerful - More so than regular expressions (Parsing expression grammar vs regular grammar).

High-level - Lets us define parsers using a BNF style notation.

Chomsky Hierarchy

Noam Chomsky did a lot of foundational research in formal language theory in the 1950s

$$REG \subset CF \subset CS \subset RE$$

REG Regular languages - those defined by regular expression

CF Context-free languages - no context in the syntactic rules

CS Context-sensitive languages - can have context in syntactic rules

RE Recursively enumerable languages - all languages that can be parsed on a computer

Parsing Expression Grammar

Pyparsing utilizes *Parsing Expression Grammars* which have been around since 2004.

Very similar to context-free languages but the choice operator | is deterministic.

Pyparsing is Pythonic

Consider the below points from "The Zen of Python":

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Readability counts.

The Zen of Pyparsing (by Paul McGuire) Part 1

Don't clutter up the parser grammar with whitespace, just handle it! (likewise for comments)

Grammars must tolerate change, as grammar evolves or input text becomes more challenging.

Grammars do not have to be exhaustive to be useful.

The Zen of Pyparsing (by Paul McGuire) Part 2

Simple grammars can return lists; complex grammars need named results.

Class names are easier to read and understand than specialized typography.

Parsers can (and sometimes should) do more than tokenize.

Arthmetic Example in Pyparsing

See arthmetic example **sum.py**

Getting Started

Loading the module inside a Python script can be done like so

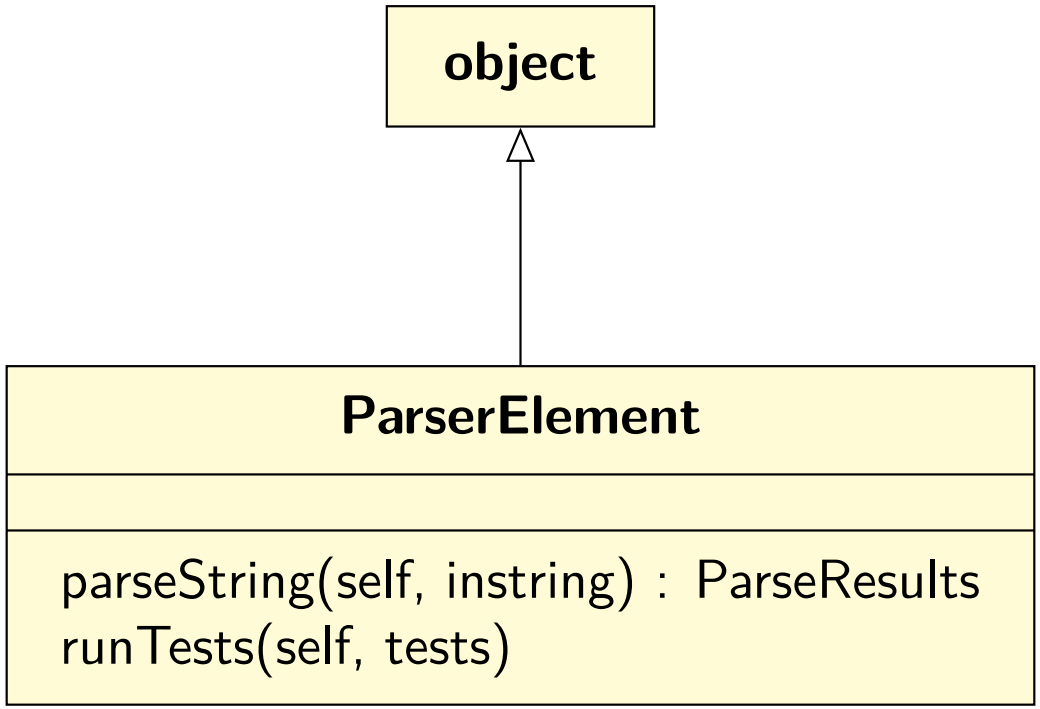
```
import pyparsing as pp
```

You can then define some parser and call either parseString or scanString.

```
myparser.parseString("Somestring")
```

Parser Element

The basic building block from which all other parsers are built



parseString

Applying the parser to a string is done using the `parseString` method.

```
sum.parseString("4 + 2 * 4")
```

```
=>
```

```
['4', '+', '2', '*', '4']
```

It should be noted that whitespace is skipped by default.

runTests

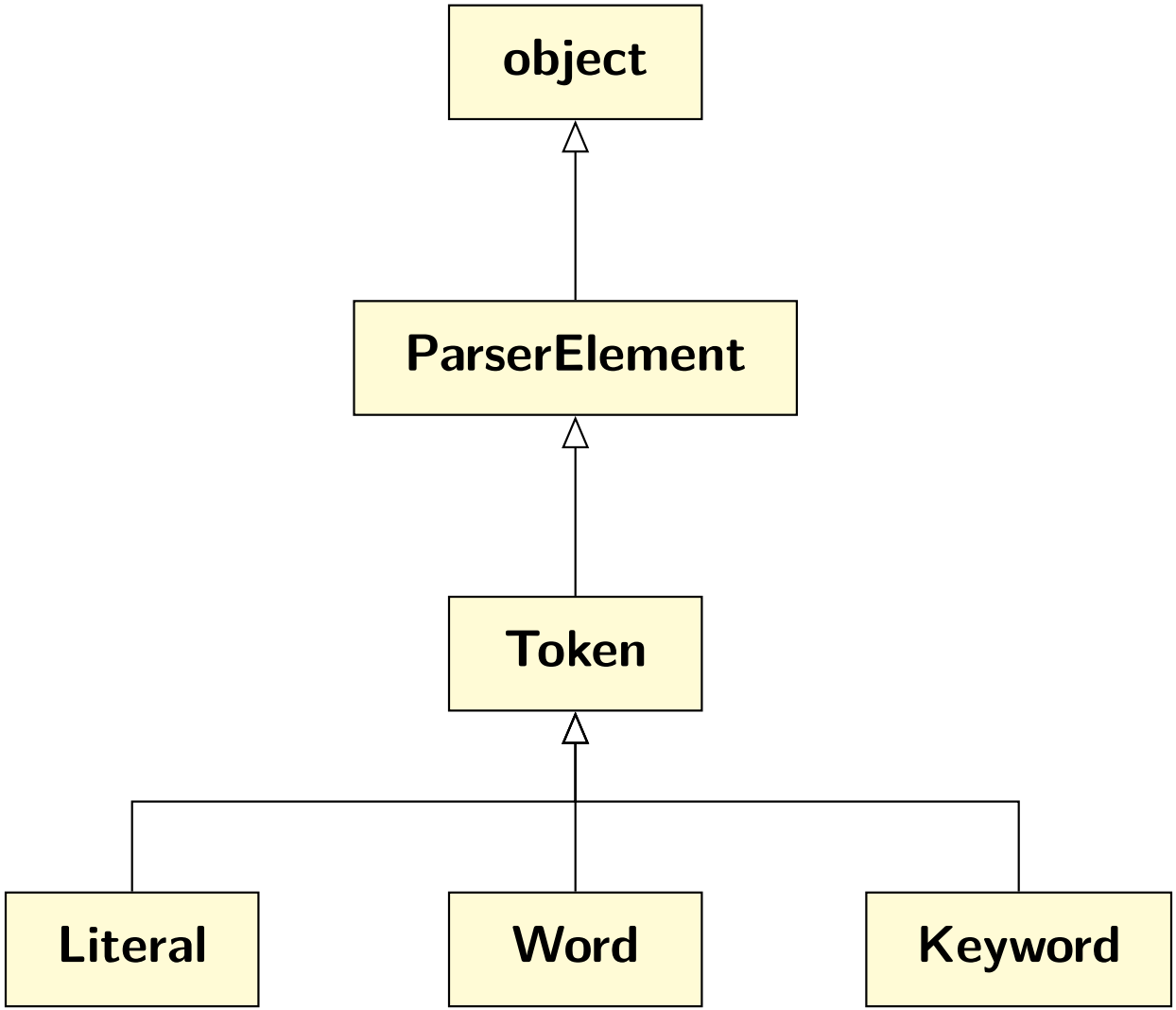
It is possible to run some simple tests using the `runTests` method.

```
tests = """
    1+2
    1+3+7
    -1*43
    -1+16*4
    """
sum.runTests(tests)
```

The results will either show that the parse succeeded or where it failed.

It is always advisable to write unit tests for real application development.

Pyparsing Basics: Defining Tokens



Pyparsing Basics: Literal

`Literal` can be used to exactly match a specified string

```
Literal('blah').parseString('blah')
     # -> ['blah']
Literal('blah').parseString('blahfooblah')
     # -> ['blah']
Literal('blah').parseString('bla')
     # -> Exception: Expected "blah"
```

There is also a `CaselessLiteral` token class for case-insensitive matching.

Pyparsing Basics: Word

Token to match a word based on a provided character set.

The following grammar
$\langle abstring \rangle ::= (\mathbf{a} \mid \mathbf{b} \mid \mathbf{A} \mid \mathbf{B})^*$

is equivalent to this Python code

```
abstring = pp.Word("abAB")
```

Pyparsing Basics: Word

There are several helper strings for defining words

helper string	description
alphas	alphabetic characters
nums	numbers
alphanums	alphabetic characters + numbers
hexnums	hexidecimal numbers
alphas8bit	alphabetic characters in ASCII range 128-255
punc8bit	non-alphabetic characters in ASCII range 128-255
printables	any non-whitespace character

Pyparsing Basics: Word

In our arthmetic example we define `unsignedint` using the `Word` Token.

```
unsignedint = pp.Word(pp.nums)

unsignedint.parseString("123")
        # => ['123']

unsignedint.parseString("abc")
# => pyparsing.ParseException: Expected W:(0123...)
```

See example **integer-withoutnames.py**

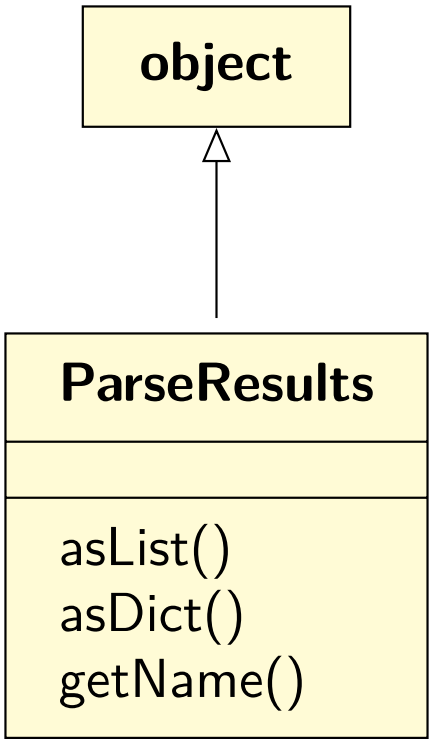
Pyparsing Basics: Word Length

Word has some helpful parameters for lengths.

exact	Match with exactly this length
max	Match with a maximum length
min	Match with a minimum length

See example **digit.py**

Pyparsing Basics: ParseResults



Pyparsing Basics: Result Names

It is possible to label results using a name

```
unsignedint = \
 pp.Word(pp.nums).setResultsName('Unsigned Integer')

unsignedint.runTests("""
        1
""")
  =>
1
['1']
- Unsigned Integer: '1'
```

See example **integer-withnames-v1.py**

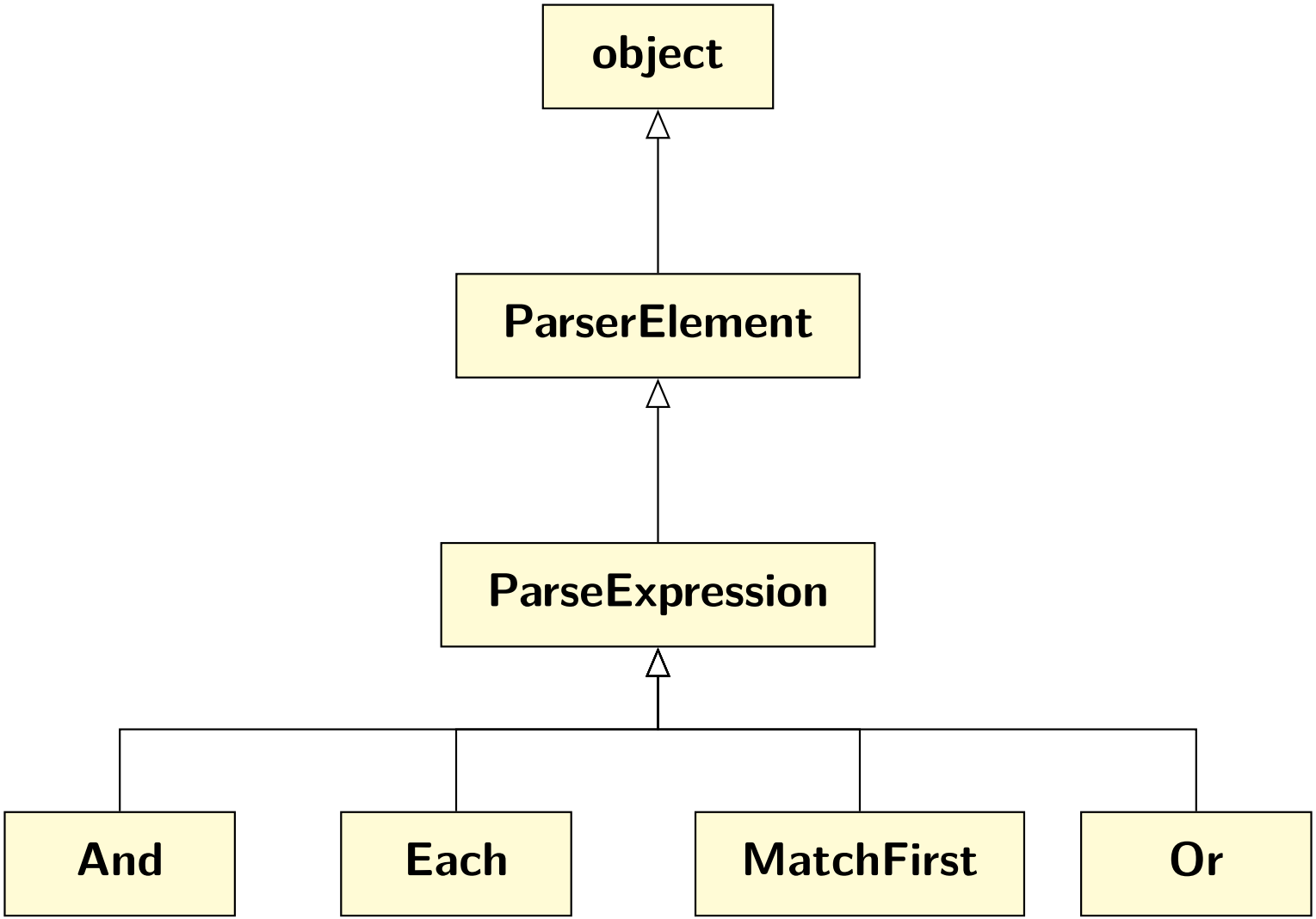
Pyparsing Basics: Keyword

The Keyword parser can be used to match works that must be seperated by white space.

```
print(pp.Keyword("if").parseString("if (1 + 1 = 2)"))
=>
["if"]
print(pp.Keyword("if").parseString("ifAndOnlyIf"))
=>
  ParseException
```

See example **ifstatement.py**

Pyparsing Basics: ParseExpressions



Pyparsing Basics: And

The `And` parse expression, written infix as +, requires that the given ParseExpressions are found in order.

```
pp.Literal("-") + unsignedint
```

or as prefix notation

```
pp.And([pp.Literal("-"), unsignedint])
```

Pyparsing Basics: Each

The `Each` parse expression, written infix as `&`, requires that the given ParseExpressions are matched, but in **any** order.

```
animal_type = pp.oneOf("CAT DOG HORSE FISH RAT")
type_attr = "type:" + animal_type("type")

name = pp.Word(pp.alphas)
name_attr = "name:" + name("pet name")

pet_spec = name_attr & type_attr
```

See example **animals-each.py**

Pyparsing Basics: Each

```
pet_spec.runTests('''
    name: Brian type: DOG
    type: CAT name: Tom
    ''')
        =>
name: Brian type: DOG
['name:', 'Brian', 'type:', 'DOG']
- pet name: 'Brian'
- type: 'DOG'

type: CAT name: Tom
['type:', 'CAT', 'name:', 'Tom']
- pet name: 'Tom'
- type: 'CAT'
```

Pyparsing Basics: MatchFirst

The `MatchFirst` parse expression, written infix as `|`, requires that one of the ParseExpressions are matched with priority given from left to right.

```
unsignedint = pp.Word(pp.nums)
integer =
    pp.Combine(
      pp.Literal("-") + unsignedint
    ).setResultsName("integer") |
        unsignedint("unsigned integer")
```

See example **integer-withnames-v2.py**

Pyparsing Basics: MatchFirst

```
integer.runTests("""
        1
        -1
""")

1
['1']
- unsigned integer: '1'

-1
['-1']
- integer: '-1'
```

Pyparsing Basics: Or

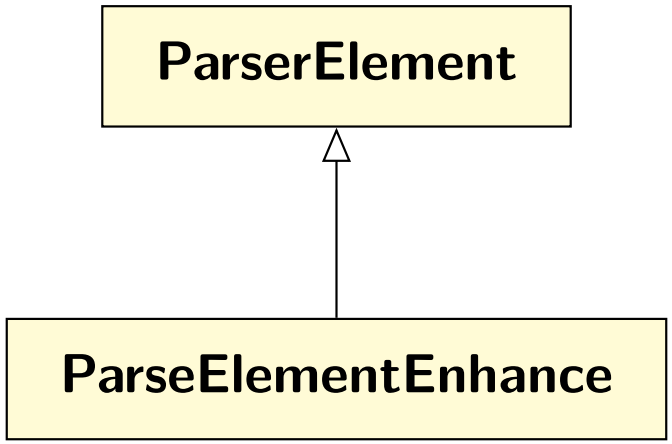
The `Or` parse expression, written infix as `^`, requires that at least one of the ParseExpressions are matched with priority given to the longest.

```
number = Word(nums) ^ Combine(Word(nums) \
         + '.' + Word(nums))
number.searchString("123 3.1416 789"))
         =>
    [['123'], ['3.1416'], ['789']]
```

See example **numberor.py**

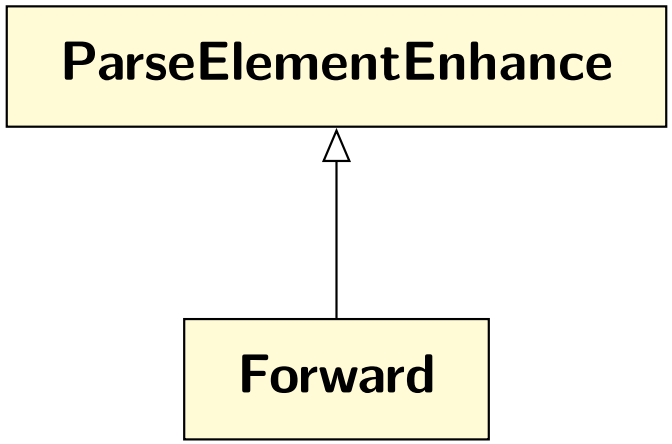
Pyparsing Basics: Parser Element Enhancements

ParseElementEnhance provides an interface for parser enhancements which combine or post process tokens.



Pyparsing Basics: Forward

The first parser enhancement we will look at is `Forward`



Pyparsing Basics: Forward

How do we handle recursion in Pyparsing? We cannot directly define `sum` like this:

$\langle sum \rangle ::= \langle sum \rangle + \langle product \rangle \mid \langle product \rangle$

Instead we must forward declare sum and use an auxilary parser:

```
sum = pp.Forward()
sum_list = product + pp.Literal("+") + sum
sum << (sum_list | product)
```

See example **sum.py**

Exercise 1: Dice Rolling

For our first exercise we will implement a dice roll parser.

For example, the roll **2d6** can be interpretted as rolling two six sided dice.

Task 1) Complete the grammar
$\langle diceroll \rangle ::= \ ...$

Exercise 1: Dice Rolling

For our first exercise we will implement a dice roll parser.

For example, the roll **2d6** can be interpretted as rolling two six sided dice.

Task 1) Complete the grammar

$\langle diceroll \rangle ::= \ ...$

Exercise 1: Dice Rolling

Discussion...

Exercise 1: Dice Rolling

One possible answer

$\langle digit \rangle ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$

$\langle numsides \rangle ::= \langle digit \rangle$

$\langle numdice \rangle ::= \langle digit \rangle$

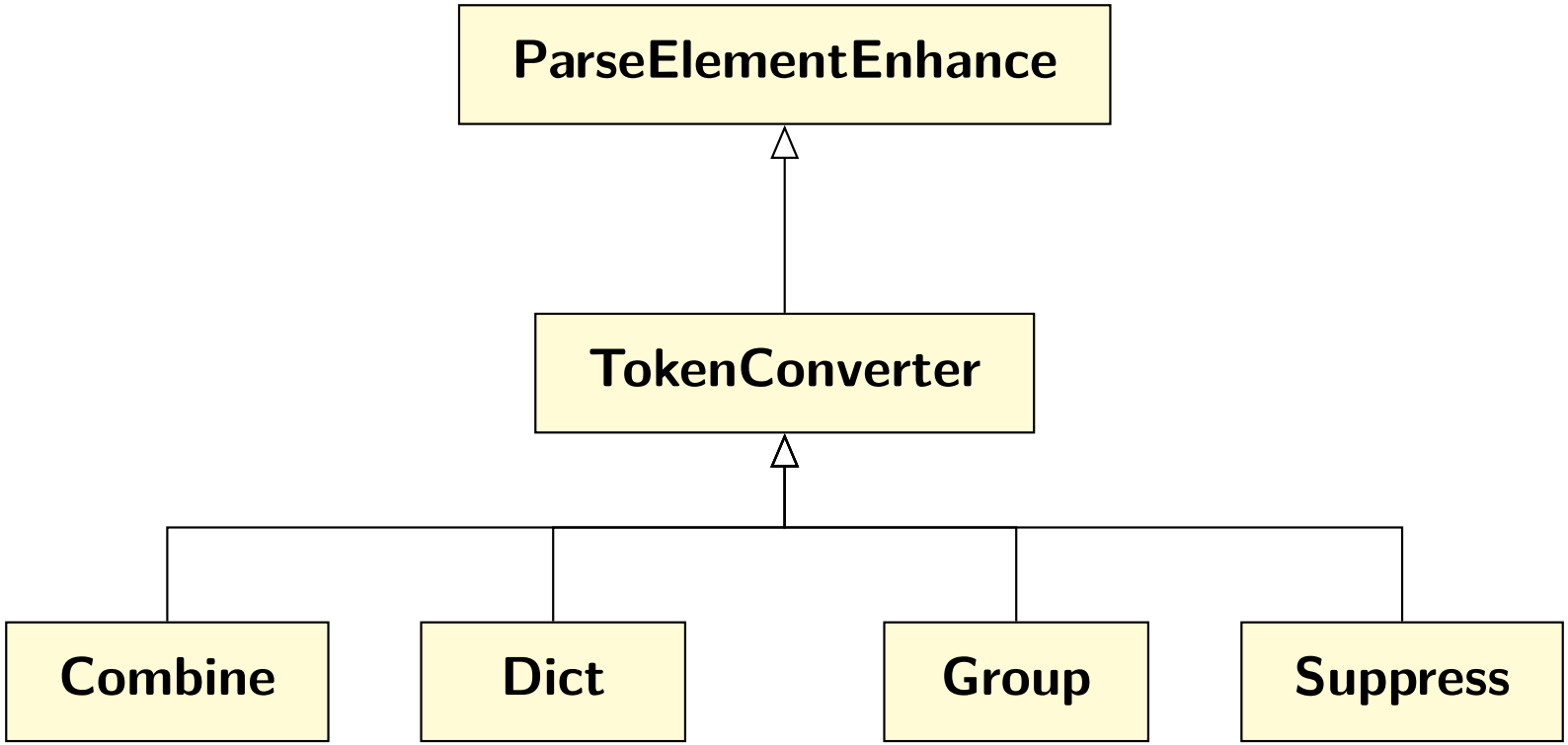
$\langle diceroll \rangle ::= \langle numdice \rangle \mathrm{d} \langle numsides \rangle$

Exercise 1: Dice Rolling

Task 2) Now implement the parser in Python. You can find a template file with some tests in the exercises folder

Pyparsing Basics: Token Converters

Token converters are used to convert parsed results



Intermediate Pyparsing: Combine

The `Combine` parse expression combines parsed tokens into a single string.

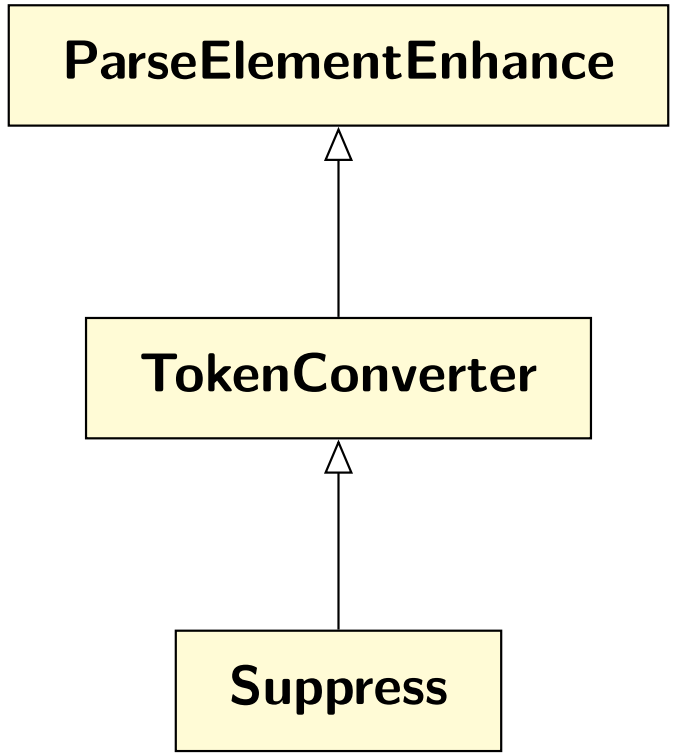
```
number = Word(nums) ^ Combine(Word(nums) + '.' + Word(nums))
number.searchString("123 3.1416 789"))
    =>
[['123'], ['3.1416'], ['789']]
```

Combine is actually enhancing the parser rather than chaining the grammar to be parsed.

See example **numberor.py**

Intermediate Pyparsing: Suppress

Suppress is a TokenConverter that throws away the parsed results



Intermediate Pyparsing: Suppress

There are two ways to apply Suppress:

```
animal_type = pp.oneOf("CAT DOG HORSE FISH RAT")
type_attr = pp.Suppress("type:") + animal_type("type")

name = pp.Word(pp.alphas)
name_attr = pp.Literal("name:").suppress() + name("pet name")

pet_spec = name_attr & type_attr
```

See example **animals-suppress.py**

Exercise 1: Dice Rolling

Task 3) Work out which tokens in your Dice Rolling Parser should be suppressed and remove them from the results.

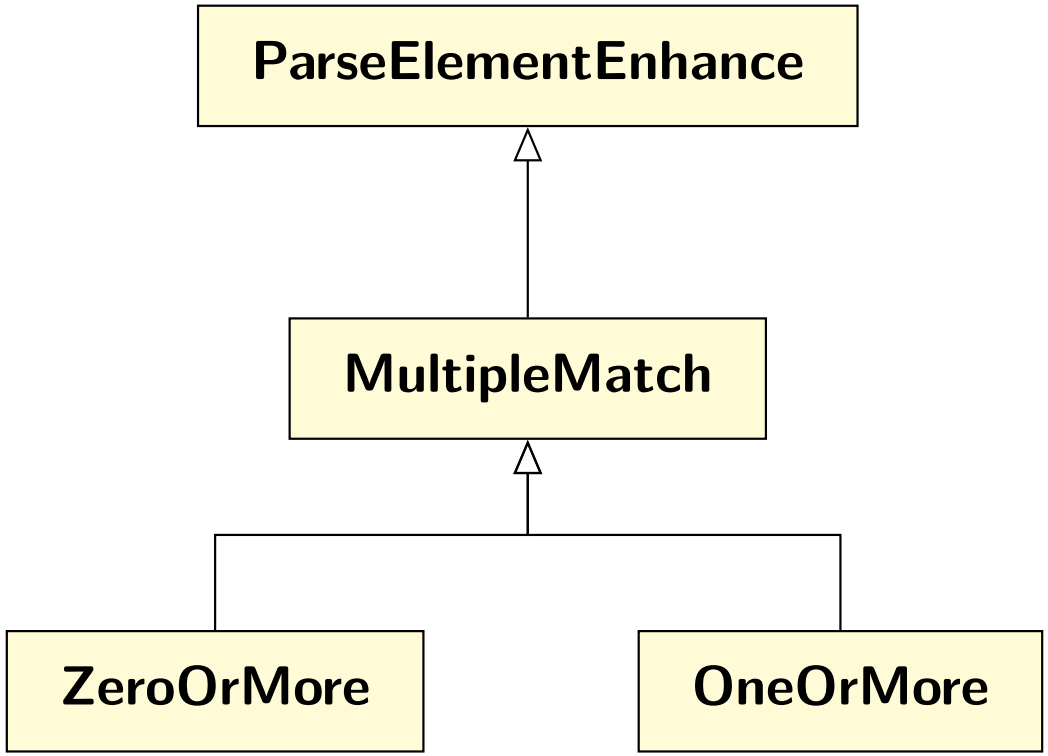
Intermediate Pyparsing: Group

It is possible to group results together into a list using the `Group` token converter.

See example **group.py**

Intermediate Pyparsing: Repetition

There are two parser enhancements to handle repetition.



`OneOrMore` was in the **group.py** example.

Intermediate Pyparsing: Adding Behavior to Parsers

Parser Element lets us add additional parsing behavior with the `setParseAction` method.

See example **sum-version2.py**.

setFailureAction

We can also add additional failure behaviour with `setFailureAction` with a function `fn(s,loc,expr,err)`.

s	The string currently being parsed
loc	The location where the failure occured
expr	The parse expression that failed
err	The exception thrown.

See example **sum-version2.py**.

Exercise 1: Dice Rolling Task 4

Task 4 Add a parse action to your dice roll parser that does a dice roll.

You may need to import the `random` module to do this.

Intermediate Pyparsing: Dict

The `Dict` token converter lets us convert tokens into Python dictionaries.

See **dictionaries.py**

Infix notation

Remember we defined a grammar for arithmetic expressions earlier...

```
sum = pp.infixNotation(pp.Word(pp.nums), [
      ("-", 1, pp.opAssoc.RIGHT),
      ("*", 2, pp.opAssoc.LEFT),
      ("+", 2, pp.opAssoc.LEFT),
    ])
```

See example **sum-version3.py**

Intermediate Pyparsing: Delimited lists

We can handle delimited lists using the `delimitedList` helper function.

```
pp.delimitedList(pp.Word(pp.nums), ',').parseString("1,2,3")
=>
['1', '2', '3']
```

See example **delimitedlist.py**

Exercise 2: JSON Parser

For our second exercise we will write a parser for a simplified JSON (JavaScript Object Notation).

This is a commonly used format for exchanging data.

What is JSON?

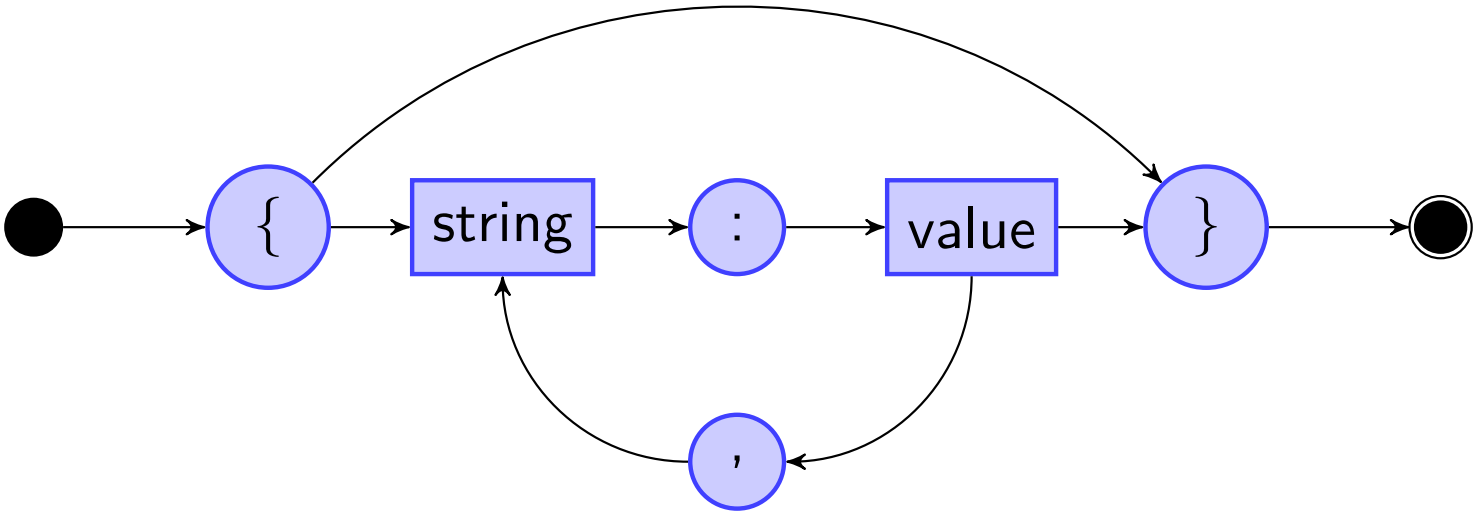
JSON is recursively from two types of datastructure can be interleaved:

- a collection of name value pairs (similar to a Python dictionary), called an **object**.
- a list of values, called an **array**

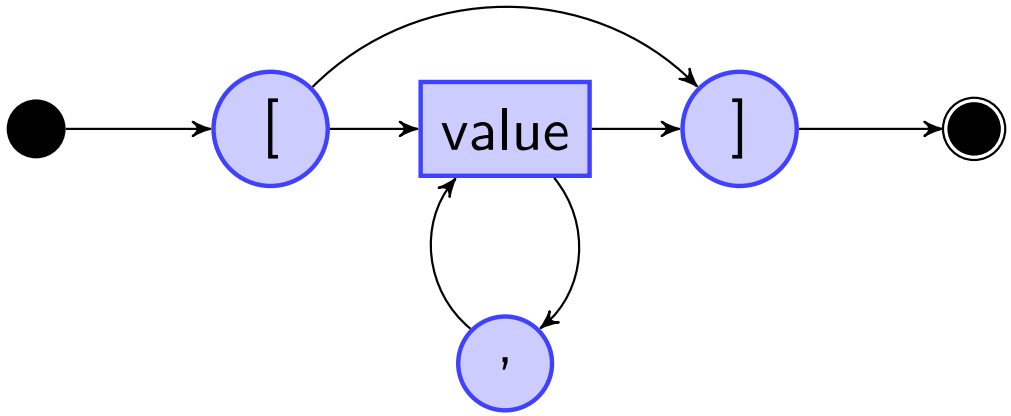
Example below

```
{ name: "John", age: 31, city: "New York" }
```

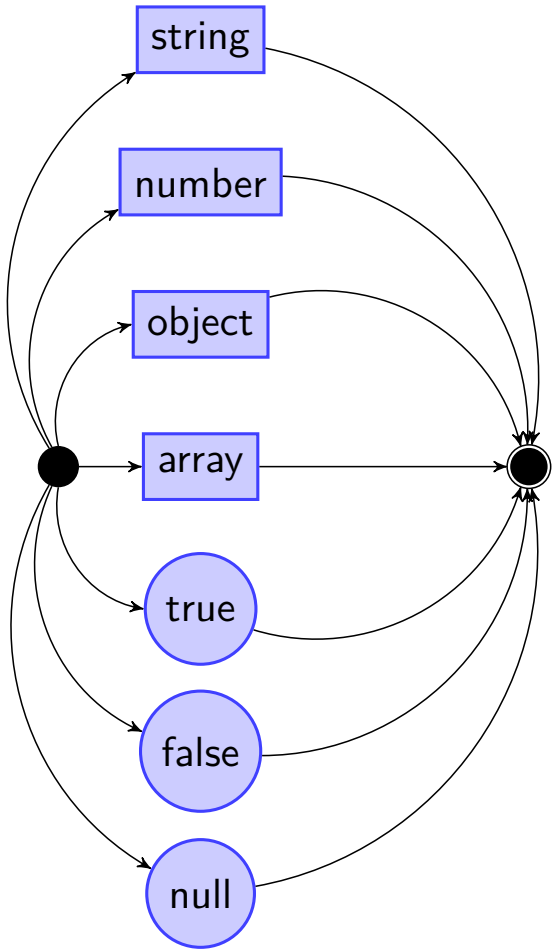
JSON Object



JSON Array

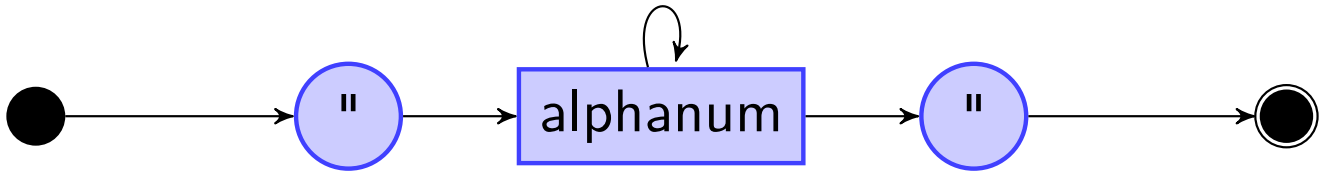


JSON Value



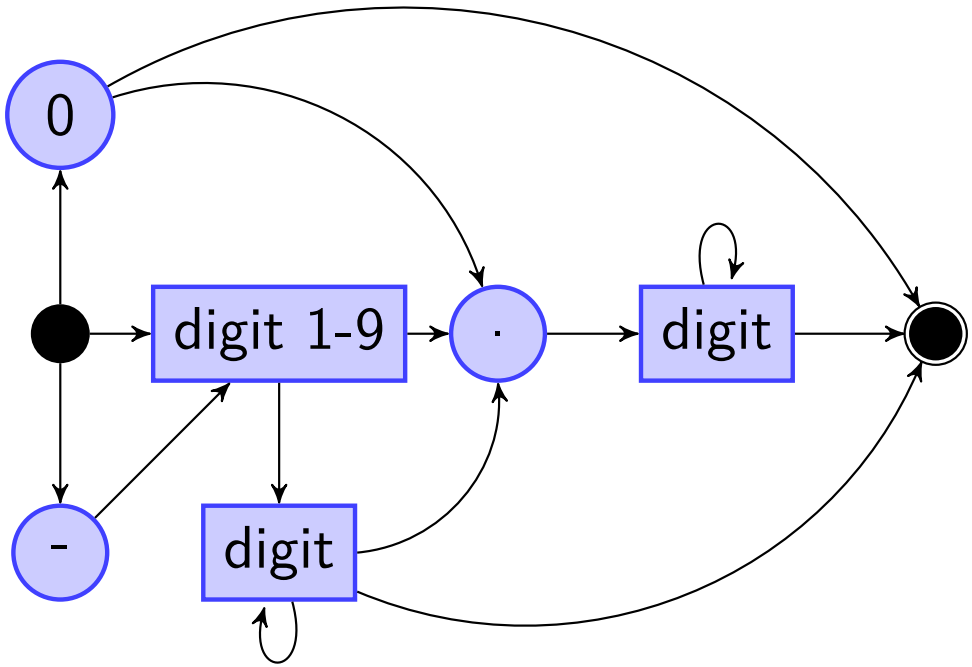
JSON String

We will use a heavily simplified version of the string for now.



JSON Number

We also use a heavily simplified version of the number.



Exercise 2 Create a JSON Parser: Task 1

Create a parser for JSON **strings** using the following grammar:

⟨*string*⟩ ::= `<alphanum>`

Exercise 2 Create a JSON Parser: Task 2

Create a parser for JSON **numbers** using the following grammar:

⟨*number*⟩ ::= ⟨*int*⟩ ⟨*frac*⟩

⟨*int*⟩ ::= ⟨*digit*⟩ | ⟨*onenine*⟩ ⟨*digits*⟩ | - ⟨*digit*⟩ | - ⟨*onenine*⟩⟨*digits*⟩

⟨*frac*⟩ ::= | . ⟨*digits*⟩

⟨*digits*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*digits*⟩

⟨*digit*⟩ ::= 0 | ⟨*onenine*⟩

⟨*onenine*⟩ ::= 1-9

Exercise 2 Create a JSON Parser: Task 3

Create a parser for JSON **values** using the following grammar:

$\langle value \rangle ::= \langle object \rangle \mid \langle array \rangle \mid \langle string \rangle \mid \langle number \rangle \mid \text{true} \mid \text{false} \mid \text{null}$

Exercise 2 Create a JSON Parser: Task 4

Create a parser for JSON **arrays** using the following grammar:

$\langle array \rangle ::= [] \mid [\ \langle values \rangle\ ]$

$\langle values \rangle ::= \langle value \rangle \mid \langle value \rangle\ ,\ \langle values \rangle$

Exercise 2 Create a JSON Parser: Task 5

Create a parser for JSON **objects** using the following grammar:

$\langle object \rangle ::= \{ \} \mid \{ \langle members \rangle \}$

$\langle members \rangle ::= \langle member \rangle \mid \langle member \rangle , \langle members \rangle$

$\langle member \rangle ::= \langle string \rangle : \langle value \rangle$

Dealing with Whitespace

By default Pyparsing parsers automatically consume whitespace.

It is possible to change this by using the `setDefaultWhitespaceChars` method e.g.

```
ParserElement.setDefaultWhitespaceChars("\t")
```

See example **whitespace.py**.

Exercise 2 Create a JSON Parser: Task 6

Extend our JSON string parser to deal with escaped characters `\t` and `\n`.

Packrat Parsing

An optimization technique for parsers. Disabled by default.

```
ParserElement.enablePackrat()
```

How does this work? It uses a technique called **memoization** or **tabling**.

Debugging

You can call `setDebug` on a parser to enable debug tracing.

See exampe **integer-debug.py**

Tracing Parse Actions

You can decorate parse actions with a tracing function.

```
@pp.traceParseAction
def dummyAction(toks):
     return None

unsignedint = pp.Word(pp.nums).setParseAction(dummyAction)
```

See example **traceparseaction.py**

Thats the end of the workshop. Thanks for listening.