

PARSING

Sid Chi-Kin Chau

[Lecture 10]



Unstructured Data



The image shows a Wikipedia page for "Parsing". The page title is "Parsing" and it is a redirect from "Parse". The content discusses the process of analyzing strings of symbols, mentioning formal grammar, sentence parsing, computational linguistics, psycholinguistics, and string analysis. A screenshot of a browser developer tools interface is overlaid on the page, specifically the Elements tab which displays the HTML structure of the page's content.

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Parsing

From Wikipedia, the free encyclopedia

"Parse" redirects here. For other uses, see [Parse \(disambiguation\)](#).

Parsing, **syntax analysis**, or **syntactic analysis** is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term *parsing* comes from Latin *pars* (*orationis*), meaning part (of speech).^[1]

The term has slightly different meanings in different branches of linguistics and computer science. Traditional sentence parsing is often performed as a method of understanding the exact meaning of a sentence or word, sometimes with the aid of devices such as sentence diagrams. It usually emphasizes the importance of grammatical divisions such as **subject** and **predicate**.

Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a [parse tree](#) showing their syntactic relation to each other, which may also contain [semantic](#) and other information.^[citation needed] Some parsing algorithms may generate a [parse forest](#) or list of parse trees for a [syntactically ambiguous](#) input.^[2]

The term is also used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc."^[1] This term is especially common when discussing what linguistic cues help

Elements Network Sources Timelines Storage Graphics Audit Console

```
grammar</div>
  <p>
    <b>Parsing</b>
    " "
    <b>syntax analysis</b>
    ", or "
    <b>syntactic analysis</b>
    " is the process of analyzing a "
    <a href="/wiki/String_(computer_science)" title="String (computer science)">string</a>
    " of "
    <a href="/wiki/Symbol_(formal)" title="Symbol (formal)">symbols</a>
    ", either in "
    <a href="/wiki/Natural_language" title="Natural language">natural language</a>
    " "
```

Style Attribute {

address, article, User Agent Style Sheet aside, div, footer, header, hgroup, layer, main, nav, section {

display: block;

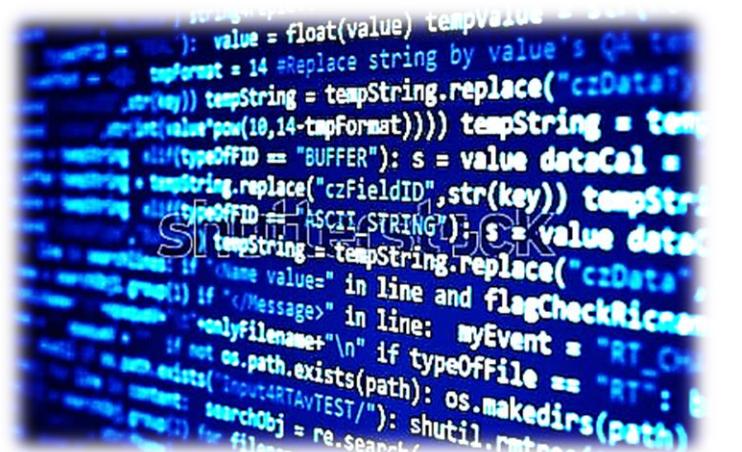
Inherited From div#mw-content-text.mw-content-ltr

Filter Classes



Goals of This Lecture

- Motivation
 - Learn how source codes are interpreted, compiled, and executed
- Tokenization
- Parsing
 - Context-free Grammars
 - Recursive Descent Parsing



Parsing Text Data

- Modern data is often stored in text files
- Input data to computer via various text files
 - Markup languages: HTML, JSON, XML, Markdown...
 - Programming languages: Java, C, C++, Haskell, Perl, Python, PHP, ...
 - Mathematical expressions, e.g., $\{(2+3)*4\}/2$
- Need to extract meaningful information for computers
- Need rules for writing and reading

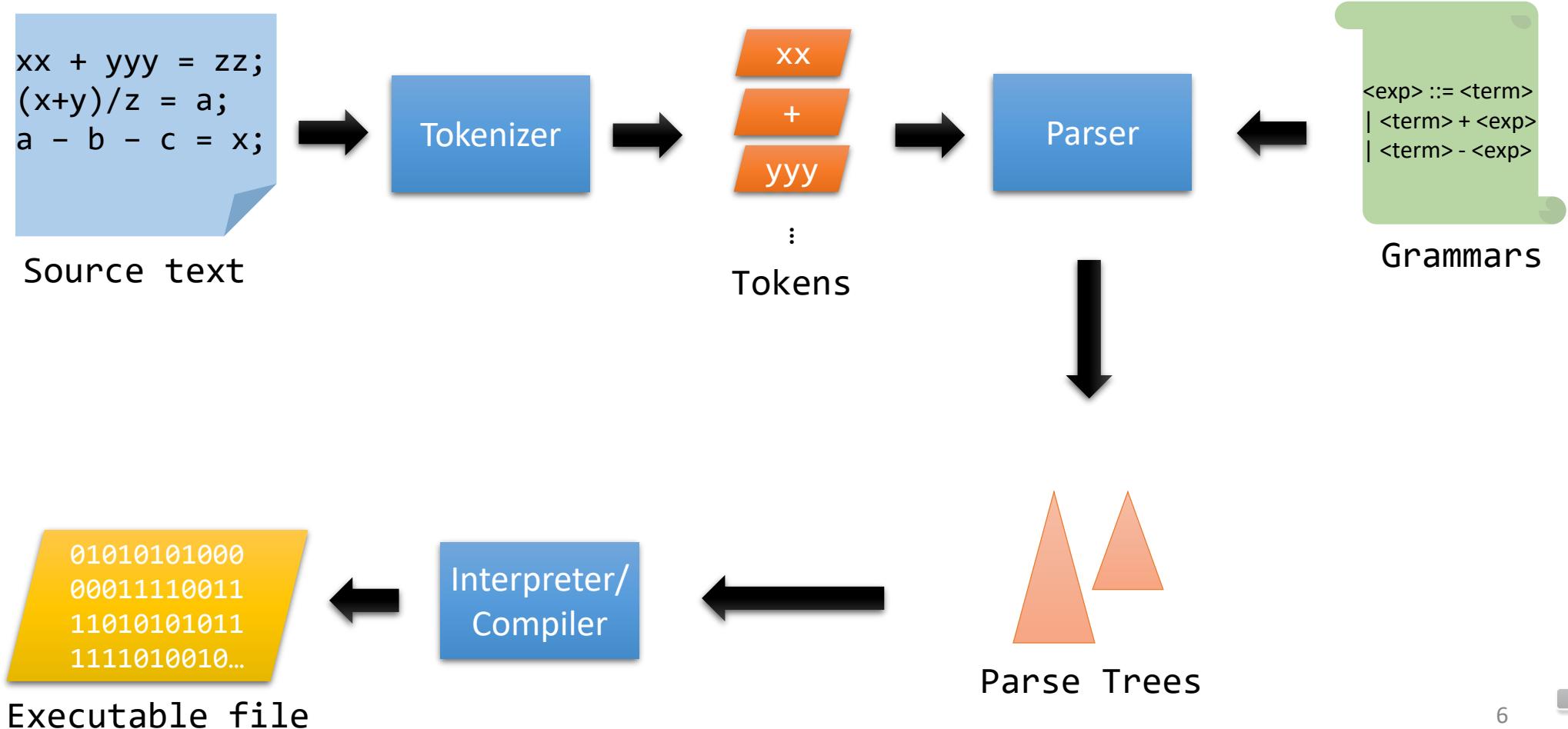


Basic Idea of Parsing

- Also called syntax (syntactic) analysis
- Aim to understand the exact meaning of structured text
 - Resulting in parse tree, a representation of structured text
 - Preceded by a tokenizer
- Need a grammar to generate parse tree
 - Grammar is a set of rules of language in structured text
 - Test whether a text conforms to the given grammar



Parsing Process



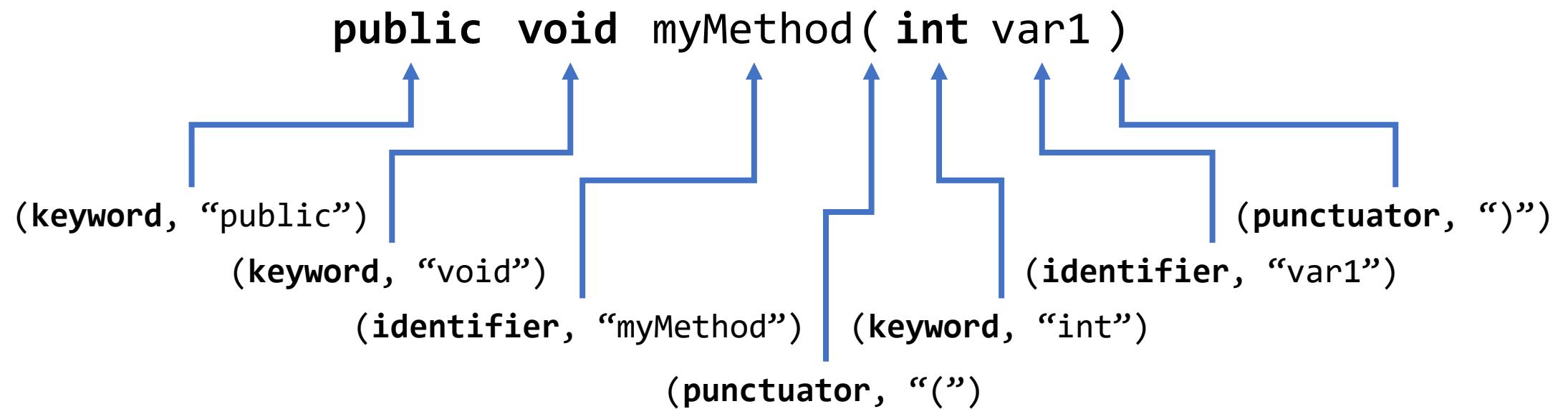
Tokenization

- Tokenization is the process of converting a sequence of characters into a sequence of tokens
- Example, natural language tokenization
 - Input
 - *"I want to tokenize this sentence."*
 - Output
 - *I / want / to / tokenize / this / sentence /.*
 - In this case, each token is a vocabulary word or a punctuation mark



Tokenization of Structured Text

- A token is a string with an assigned meaning as a pair consisting of a token type and a token value
- Example:



Token Types

- Tokens: type, location, name (if any)

Token Type	Example
Punctuators	() ; , []
Operators	+ - * :=
Keywords	begin end if while try catch
Identifiers	Square_Root
String literals	“press Enter to continue”
Character literals	'x'
Numeric literals (integer)	123
Numeric literals (floating point)	5.23e+2



Punctuators (Separators)

- Typically individual special characters
 - Example: ({ } : . ;
 - Sometimes double characters: tokenizer looks for longest token:
 - /*, //, -- comment openers in various languages
 - Returned as identity (type) of token
 - Plus perhaps location for error messages and debugging purposes



Operators

- Like punctuators
 - No real difference for tokenizer
 - Tokenizers do not “process/execute” tokens
 - Typically single or double special characters
 - Operators
 - + - == <=
 - Operations
 - :=
 - Returned as type of token
 - Plus perhaps location



Identifiers and Keywords

- Identifiers: function names, variable names
 - Length, allowed characters, separators
- Need to build a name table
 - Single entry for all occurrences of names like var1, myFunction
 - Typical data structure: hash table
- Tokenizer returns token types
 - Plus key (index) to table entry
 - Table entry includes location information
- Keywords: Reserved identifiers (it can be case-sensitive)
 - e.g., BEGIN END in Pascal, if in C, catch in C++



Literals

- Pre-defined constants in programming languages
- String literals
 - “example”
- Character literals
 - ‘c’
- Numeric literals
 - 123 (Integer)
 - 123.456 (Double)



Free-form vs. Fixed format

- Free-form languages (modern ones)
 - White space does not matter. Ignore these:
 - Tabs, spaces, new lines, carriage returns
 - Only the ordering of tokens is important
 - e.g., C, C++, Java, Javascript, ...
- Fixed format languages (historical ones)
 - Layout is critical
 - Fortran, Python, indentation
 - Tokenizer must know about layout to find tokens
 - It was born in 1950's (for punched cards, one statement per card – easy to debug/maintain)



Case Equivalence

- Some programming languages are case-insensitive
 - Pascal, Ada
- Some programming languages are case-sensitive
 - C, Java
- Tokenizer ignores case if configured
 - `This_Routine == THIS_RouTine`
 - Error analysis of tokens may need exact cases

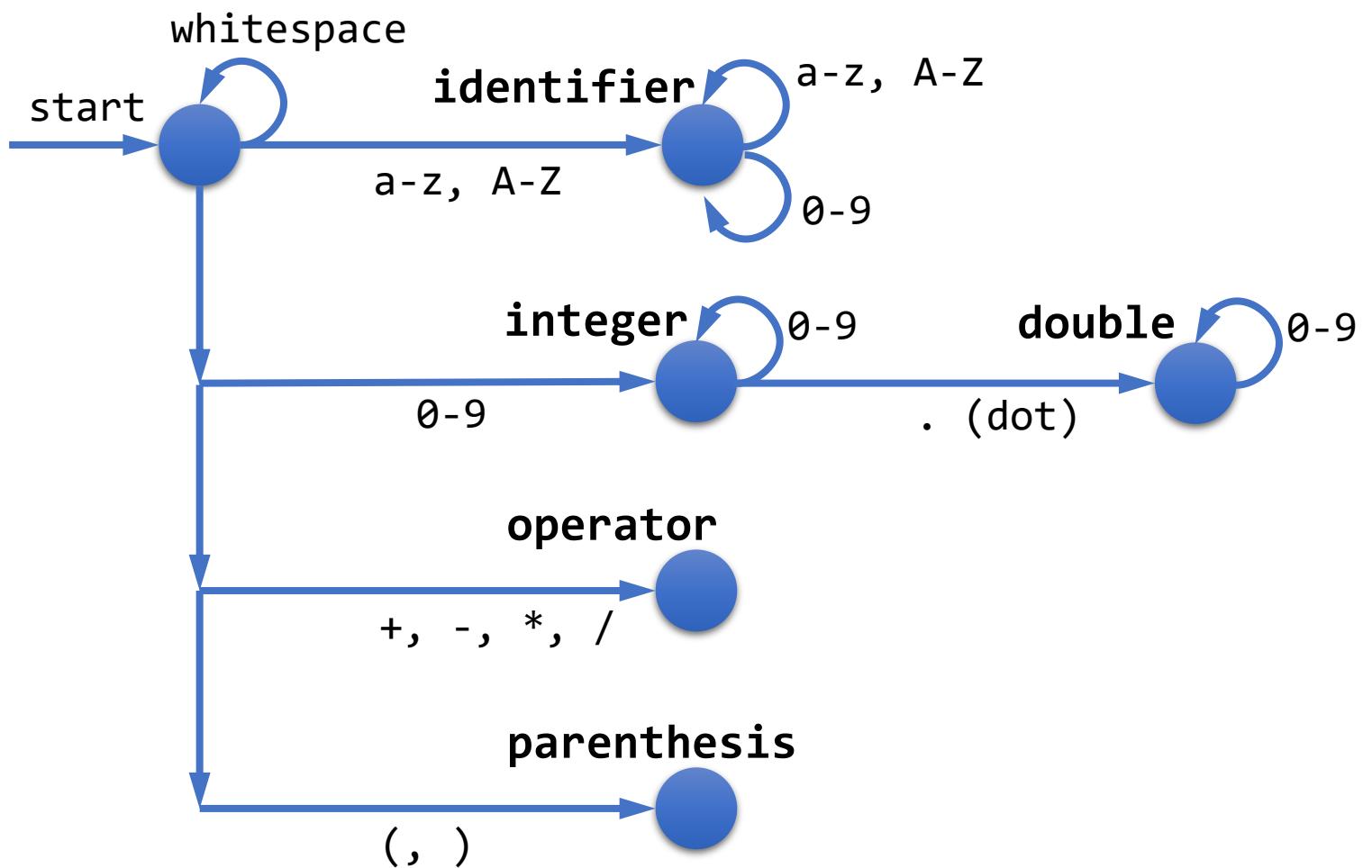


Tokenizer

- Tokenizer carries out pre-processing of the source text
- A finite-state machine (FSM) is used to track the input sequences of characters that can be contained within any recognized tokens
- The first non-whitespace character can be used to deduce the kind of token that follows
- Subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token



Finite State Machine of Tokenizer



General Strategy for Tokenization

- Tokenization aims to split the words in the source text
- Practical source text contain certain redundant information
- Tokenizer relies on simple strategies:
 - Punctuation and whitespace may or may not be included in the resulting list of tokens
 - All contiguous strings of alphabetic characters are part of one token; likewise with numbers
 - Tokens are separated by whitespace characters, such as a space or line break, or by punctuation characters



General Approach

- Define a set of token types:
 - Enumeration type
 - `tok_int`, `tok_if`, `tok_plus`, `tok_left_paren`, etc
- Tokenizer returns a pair consisting of a token name and an optional token value
 - Some tokens carry associated data
 - e.g. Location in the text
 - Key for identifier table



Abstract Class Tokenizer

```
public abstract class Tokenizer {  
  
    // extract next token from the current text and save it  
    public abstract void next();  
  
    // return the current token (without type information)  
    public abstract Object current();  
  
    //check whether there is a token remaining in the text  
    public abstract boolean hasNext();  
}
```



What is a Grammar?

- Grammar is a formal method to describe a (textual) language
- Simple solution: Defining a finite set of all acceptable sentences
 - Problems:
 - Large space/memory complexity
 - What to do with infinite languages?
- Practical solution: a finite recipe to describe all acceptable sentences
 - A grammar is a finite description of a possibly infinite set of acceptable sentences



Grammars

- Grammars formally specify how languages are constructed
 - A grammar specifies the rules of a language
 - Production rule: “ $a \rightarrow b$ ” means that a can be rewritten as b
- Example, simplified English grammar:

```
<sentence> → <noun_phrase> <predicate>
<noun_phrase> → <article> <noun>
<predicate>   → <verb>
<article> → a
<article> → the
<noun> → cat
<noun> → dog
<verb> → runs
<verb> → sleeps
```



Example

- Derivation of string “the dog sleeps” from the simplified English grammar

```
<sentence> → <noun_phrase> <predicate>
<noun_phrase> → <article> <noun>
<predicate> → <verb>
<article> → a
<article> → the
<noun> → cat
<noun> → dog
<verb> → runs
<verb> → sleeps
```

Derivation	Grammar Rule
$\langle \text{sentence} \rangle \Rightarrow \langle \text{noun_phrase} \rangle \langle \text{predicate} \rangle$	$\langle \text{sentence} \rangle \rightarrow \langle \text{noun_phrase} \rangle \langle \text{predicate} \rangle$
$\Rightarrow \langle \text{noun_phrase} \rangle \langle \text{verb} \rangle$	$\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle$
$\Rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle$	$\langle \text{noun_phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$
$\Rightarrow \text{the } \langle \text{noun} \rangle \langle \text{verb} \rangle$	$\langle \text{article} \rangle \rightarrow \text{the}$
$\Rightarrow \text{the dog } \langle \text{verb} \rangle$	$\langle \text{noun} \rangle \rightarrow \text{dog}$
$\Rightarrow \text{the dog sleeps}$	$\langle \text{verb} \rangle \rightarrow \text{sleeps}$



Example

- Derivation of string “a cat runs” from the simplified English grammar:

```
<sentence> → <noun_phrase> <predicate>
<noun_phrase> → <article> <noun>
<predicate> → <verb>
<article> → a
<article> → the
<noun> → cat
<noun> → dog
<verb> → runs
<verb> → sleeps
```

Derivation	Grammar Rule
$\langle \text{sentence} \rangle \Rightarrow \langle \text{noun_phrase} \rangle \langle \text{predicate} \rangle$	$\langle \text{sentence} \rangle \rightarrow \langle \text{noun_phrase} \rangle \langle \text{predicate} \rangle$
$\Rightarrow \langle \text{noun_phrase} \rangle \langle \text{verb} \rangle$	$\langle \text{predicate} \rangle \rightarrow \langle \text{verb} \rangle$
$\Rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle$	$\langle \text{noun_phrase} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun} \rangle$
$\Rightarrow \text{a } \langle \text{noun} \rangle \langle \text{verb} \rangle$	$\langle \text{article} \rangle \rightarrow \text{a}$
$\Rightarrow \text{a } \text{cat } \langle \text{verb} \rangle$	$\langle \text{noun} \rangle \rightarrow \text{cat}$
$\Rightarrow \text{a } \text{cat } \text{runs}$	$\langle \text{verb} \rangle \rightarrow \text{runs}$



Language of Grammar

- Language is all possible derivations from a given grammar
- Example, the language of the simplified English grammar:

$$L = \{ \text{"a cat runs"}, \\ \text{"a cat sleeps"}, \\ \text{"the cat runs"}, \\ \text{"the cat sleeps"}, \\ \text{"a dog runs"}, \\ \text{"a dog sleeps"}, \\ \text{"the dog runs"}, \\ \text{"the dog sleeps"} \}$$


Chomsky's Grammar Hierarchy

- Type-0: Recursively Enumerable
 - Specified by a Turing machine (any computer)
- Type-1: Context-sensitive
 - Rules: $\alpha A \beta \rightarrow \alpha \gamma \beta$
- Type-2: Context-free
 - Rules: $A \rightarrow \gamma$
- Type-3: Regular
 - Specified by a finite state machine



Context Free Grammars

- Grammars provide a precise way of specifying languages
- A context free grammar is often used to define the syntax
- A context free grammar is specified via a set of production rules
 - Specified by \rightarrow or $::=$
 - Variables (or non-terminals; surrounded with <>)
 - Example: <noun_phrase>, <verb>
 - Terminals (symbols)
 - Example: a, cat, runs
 - Alternatives (|)
 - Example: <article> \rightarrow a | the



Example

- Grammar for integers
 - $\{1, 2, 123, 001, 123001, 76860, 00000000, \dots\}$

```
<num> → <digit><num>|<digit>
<digit> → 0|1|2|3|4|5|6|7|8|9
```

- Exercise: What is the grammar for an unsigned integer number that is not starting with '0' character?



Exercise



- Grammar of matching parentheses:

```
<S> → <S> <S>
<S> → (<S>)
<S> → ()
```

```
<S> → <S> <S>
<S> → )<S>(
<S> → ()
```

- $()((())$: Acceptable/Not Acceptable?
- $((())()(((())$: Acceptable/Not Acceptable?
- $((())$: Acceptable/Not Acceptable?
- $))())((:$ Acceptable/Not Acceptable?
- $)())()$: Acceptable/Not Acceptable?
- $)())()()$: Acceptable/Not Acceptable?



Online Demo

- Web-based grammar generator
 - <https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/>
 - Useful for online learning



Test

To test the CFG above, input test strings here, one per line. An empty line corresponds to the empty string. Results will be shown automatically.

```
1  
20  
151
```

Test Results for CFG

#	String	Matches	
1	"1"	Yes	See Derivation
2	"20"	Yes	See Derivation
3	"151"	Yes	See Derivation



Formal Definitions

- Grammar: $G=(V, T, S, P)$
 - V : Set of variables
 - T : Set of terminal symbols
 - S : Start variable
 - P : Set of production rules
- Example
 - $V = \{\langle \text{digit} \rangle, \langle \text{num} \rangle\}$
 - $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
 - $S = \langle \text{num} \rangle$
 - $P =$

```
<num> → <digit><num> | <digit>
<digit> → 0|1|2|3|4|5|6|7|8|9
```



Grammar for Mathematical Expressions

- Mathematical expressions can be generated by the following grammar:

- $V = \{\langle E \rangle\}$
- $T = \{a, +, *, (,)\}$
- Let's assume 'a' can be any number

- $S = \langle E \rangle$

- $P = \langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle \mid \langle E \rangle * \langle E \rangle \mid (\langle E \rangle) \mid a$



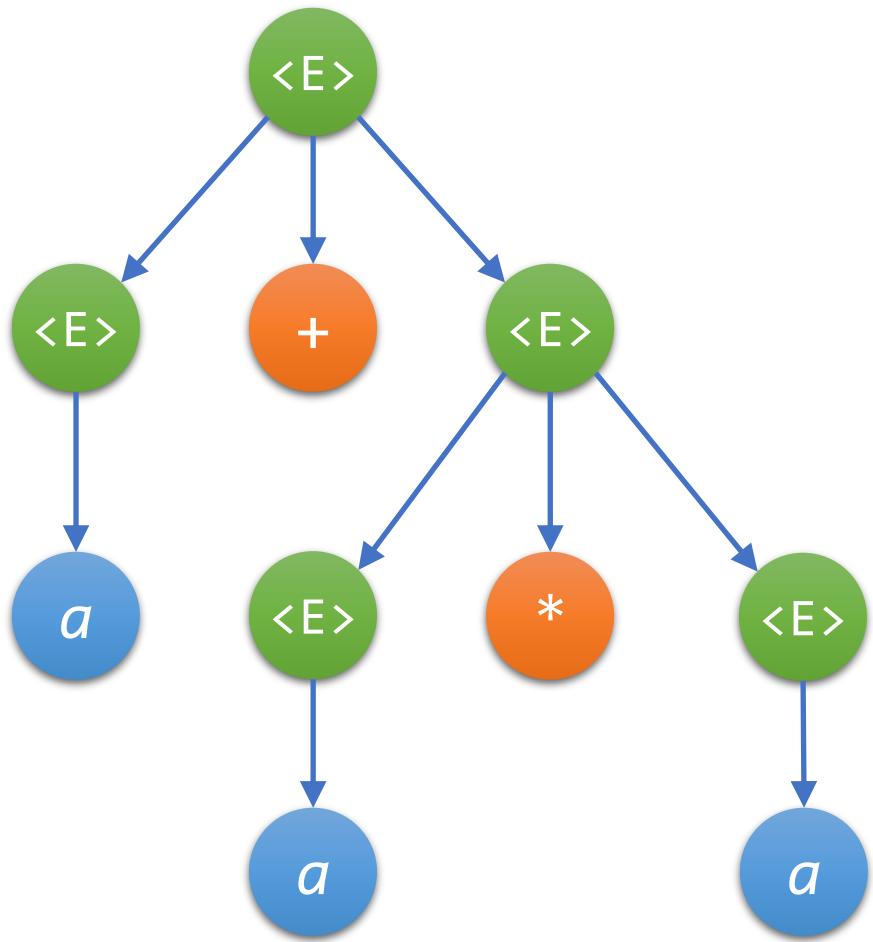
Example

- Derivation of expression “ $a + a * a$ ” from the grammar

$$\begin{aligned}\langle E \rangle &\Rightarrow \langle E \rangle + \langle E \rangle \\&\Rightarrow a + \langle E \rangle \\&\Rightarrow a + \langle E \rangle * \langle E \rangle \\&\Rightarrow a + a * \langle E \rangle \\&\Rightarrow a + a * a\end{aligned}$$

```
<E> → <E> + <E> | <E> * <E> | (<E>) | a
```

Grammar Production Rule

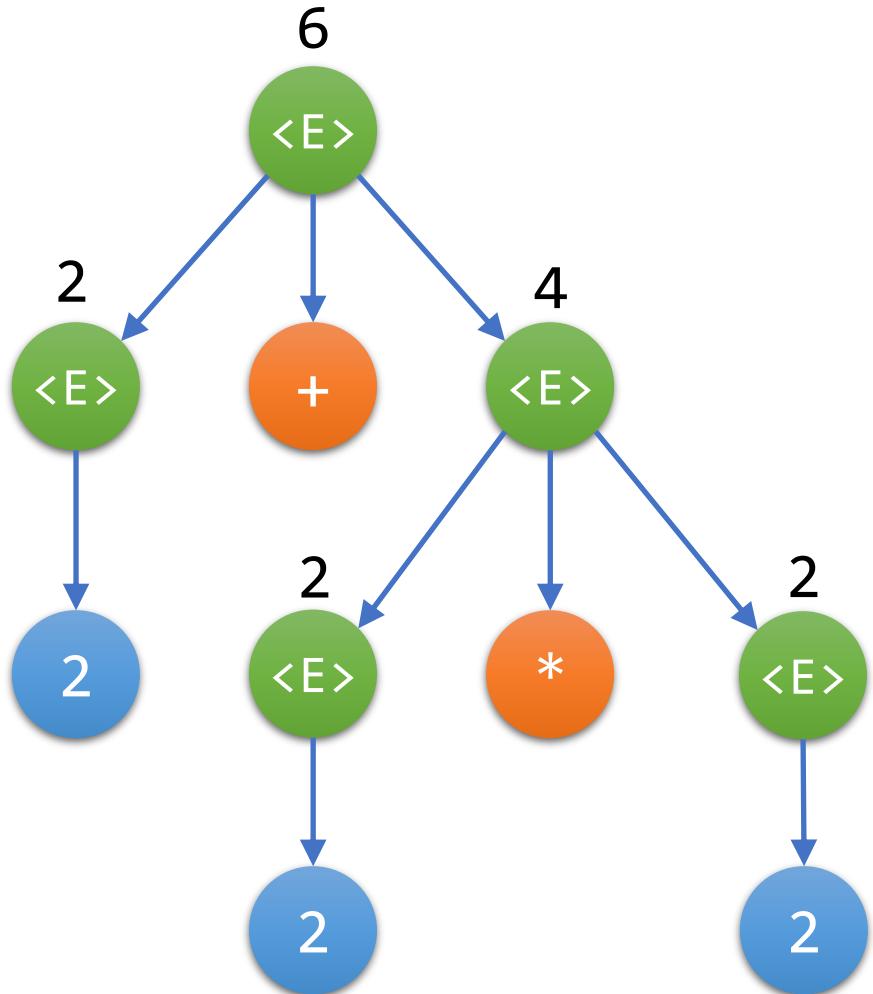


Parse Tree



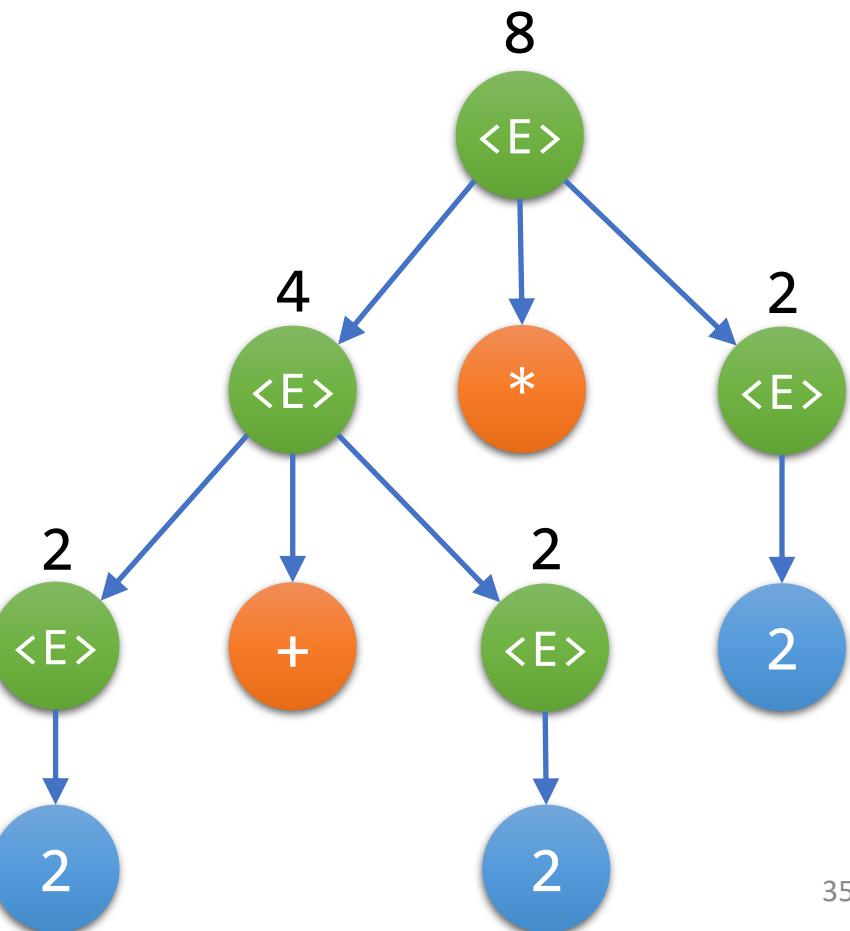
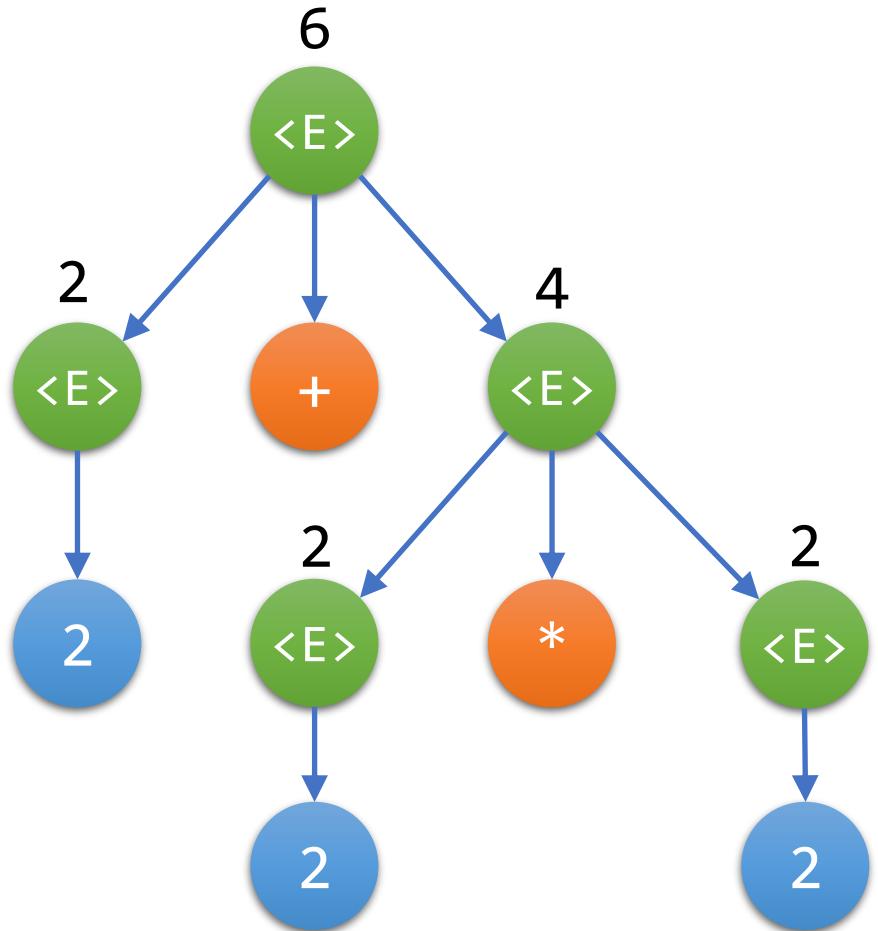
Evaluate Expression via Parse Tree

- Parsing tree defines an evaluation order
- Evaluation of expression can be evaluated from leaves to root
- Intermediate variables are replaced by terminal values iteratively



Ambiguity

- It is ambiguous because there are two possible derivations!



Ambiguous Grammar

- A context free grammar G is ambiguous
 - If there is a string w from language of grammar G which has
 - More than one derivation tree
- But it is possible to remove ambiguity by restructuring the grammar

```
<E> → <E> + <E>
<E> → <E> * <E>
<E> → (<E>) | a
```

Ambiguous Grammar

```
<E> → <T> + <E> | <T>
<T> → <F> * <T> | <F>
<F> → (<E>) | a
```

Non-ambiguous Grammar



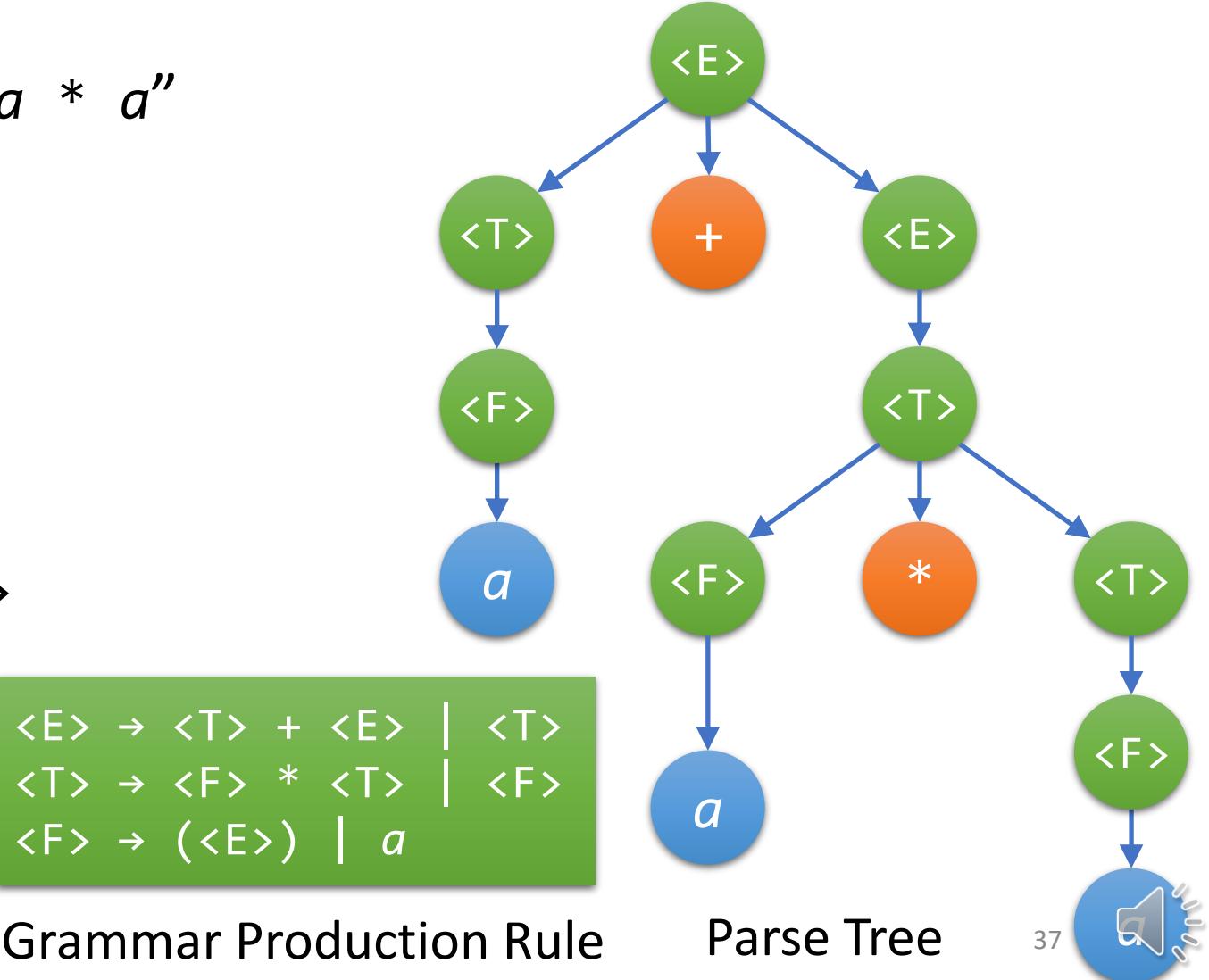
Example

- Derivation of expression “ $a + a * a$ ” from the grammar

$$\begin{aligned} <E> &\Rightarrow <T> + <E> \\ &\Rightarrow <F> + <E> \\ &\Rightarrow a + <E> \\ &\Rightarrow a + <T> \\ &\Rightarrow a + <F> * <T> \\ &\Rightarrow a + a * <T> \\ &\Rightarrow a + a * <F> \\ &\Rightarrow a + a * a \end{aligned}$$
$$\begin{array}{l|l} <E> \rightarrow <T> + <E> & <T> \rightarrow <F> * <T> \\ <T> \rightarrow <F> * <T> \mid <F> & <F> \rightarrow (<E>) \mid a \end{array}$$

Grammar Production Rule

Parse Tree



Example

- Check non-ambiguous Grammar
 - <https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/>



Test Results for CFG

#	String	Matches
1	"2+2*2"	Yes See Derivation
Rule	Application	Result
Start → S	Start	S
S → T+S	S	T+S
T → F	T+S	F+S
F → a	F+S	a+S
a → 2	a+S	2+S
S → T	2+S	2+T
T → F*T	2+T	2+F*T
F → a	2+F*T	2+a*T
a → 2	2+a*T	2+2*T
T → F	2+2*T	2+2*F
F → a	2+2*F	2+2*a
a → 2	2+2*a	2+2*2



Implementing Parser

- Recursive descent parser
 - Top-down parser
 - Parse from start variable, recursively parse input tokens
 - Create a method for each left-hand side variable in the grammar production rules
 - These methods are responsible for generating parsed nodes



Variable (or Symbol) as a node

- To construct a parse tree,
 - Can adopt ideas from a tree data structure
 - Each variable (or symbol) can be represented as a node in a tree
 - Define a node class for each variable



Implement a Parser

- Given grammar start with Exp

```
<Exp> → <Term> + <Exp> | <Term>
<Term> → <Factor> * <Term> | <Factor>
<Factor> → (<Exp>) | a
```

- We implement based on Recursive Descent Parsing
- Node classes
 - abstract class Exp
 - class Term extends Exp
 - class Factor extends Exp
 - class Int extends Exp
 - class AddExp extends Exp
 - class MulExp extends Exp
- Parse method for each rule in Parse class
 - Exp parseExp(Tokenizer)
 - Exp parseTerm(Tokenizer)
 - Exp parseFactor(Tokenizer)



Implement a Parser

```
public class Parser {  
  
    // parse <Exp> -> <Term> + <Exp> | <Term>  
    public static Exp parseExp(Tokenizer tok){ . . . }  
  
    // parse <Term> -> <Factor> * <Term> | <Factor>  
    public static Exp parseTerm(Tokenizer tok){ . . . }  
  
    // parse <Factor> -> (<Exp>) | a  
    public static Exp parseFactor(Tokenizer tok){ . . . }  
}
```



Implement a Parser

- To implement rule “ $\langle \text{Exp} \rangle \rightarrow \langle \text{Term} \rangle + \langle \text{Exp} \rangle \mid \langle \text{Term} \rangle$ ”

```
public static Exp parseExp(Tokenizer tok){  
    Term term = parseTerm(tok);  
    if (tok.current()=='+'){  
        tok.next();  
        Exp exp = parseExp(tok);  
        return new AddExp(term, exp);  
    } else {  
        return term;  
    }  
}
```

Addition between
Term and Exp

This production rule
starts with Term

If the next token is +, apply
first production rule



Implement a Parser

- To implement rule “ $\langle \text{Term} \rangle \rightarrow \langle \text{Factor} \rangle * \langle \text{Term} \rangle \mid \langle \text{Factor} \rangle$ ”

```
public static Exp parseTerm(Tokenizer tok){  
    Factor factor = parseFactor(tok);  
    if (tok.current()=='*'){  
        tok.next();  
        Term term = parseTerm(tok);  
        return MulExp(factor, term);  
    } else {  
        return factor;  
    }  
}
```

Multiplication between
Factor and Term

This production rule
starts with Factor

If the next token is *, apply
first production rule



Implement a Parser

- To implement rule “<Factor> → (<Exp>) | a”

```
public static Exp parseFactor(Tokenizer tok){  
    if (tok.current()=='('){  
        tok.next();  
        Exp exp = parseExp(tok);  
        tok.next();  
        return exp;  
    } else {  
        Int i = new Int(tok.current());  
        return i;  
    }  
}
```

If the next token is '(', apply the first production rule

Remove ')

Create an integer



Limitations

- Recursive Descent Parsing: Top-down parser from recursive procedures
 - The Recursive Descent Parsing approach will not work with left recursive grammars. For example,

```
<binary> → <binary><digit>|<digit>  
<digit> → 0|1
```

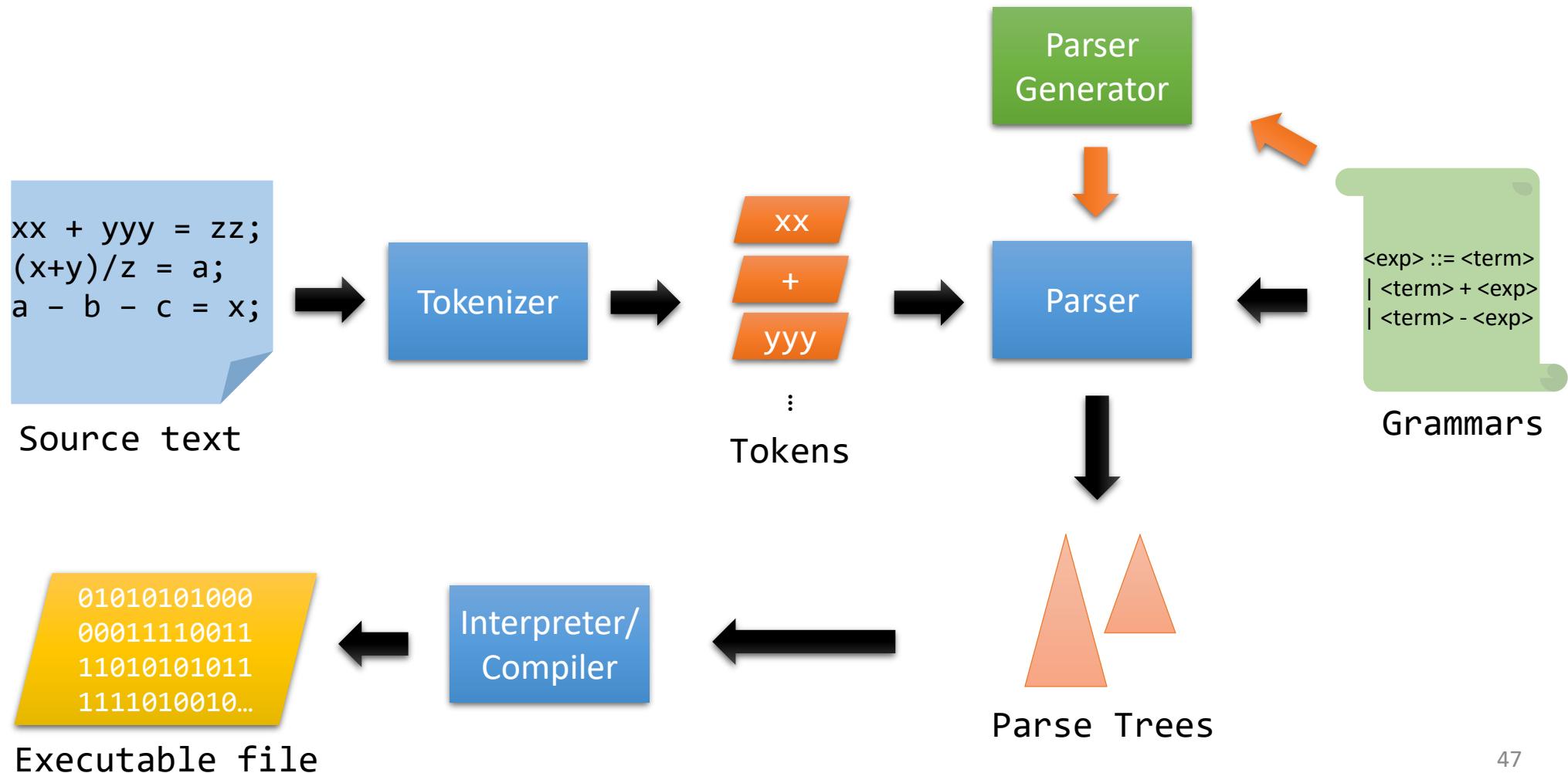
- which cannot be parsed using the recursive descent parser
- However we could transform the grammar into:

```
<binary> → <digit><binary>|<digit>  
<digit> → 0|1
```

- which represents the same language, but can be parsed by recursive descent parser



Automatic Parser Generation



Automatic Parser Generation

- Parser Generators
 - JavaCC, ANTLR, Yacc/Bison
- Manual Parser Implementation
 - Good error recovery and reporting
 - Flexible combination of parsing and actions
- Automatic Parser Generation
 - Save manual work
 - But more complex and rigid frameworks
 - Error recovery and reporting more difficult



Summary

- Structured data
 - Markup languages: HTML, JSON, XML, Markdown...
 - Programming languages: Java, C, C++, Haskell, Perl, Python, PHP, ...
 - Mathematical expressions, e.g., $\{(2+3)^*4\}/2$
- Tokenization
- Parsing
 - Context-free Grammars
 - Recursive Descent Parsing



```
#DEAR FUTURE SELF,  
#  
# YOU'RE LOOKING AT THIS FILE BECAUSE  
# THE PARSE FUNCTION FINALLY BROKE.  
#  
# IT'S NOT FIXABLE. YOU HAVE TO REWRITE IT.  
# SINCERELY, PAST SELF
```

| DEAR PAST SELF, IT'S KINDA
| CREEPY HOW YOU DO THAT.

```
#ALSO, IT'S PROBABLY AT LEAST  
# 2013. DID YOU EVER TAKE  
# THAT TRIP TO ICELAND?
```

STOP JUDGING ME!

