Week 6 Announcements

- Second-semester *census date*: Thursday, 31 August.
- Video Assignment #1 → No late submission (hard deadline).
- Group Project:
  - To be released this week.
  - Do you already have your groups (team)?
  - Team registration will be open from this week!
- This Friday, 01 September we'll have an additional lecture:
  - Tokeniser and Parser (finishing) | Group Project Tips | GitLab
- Additional (optional) homework: **Red-Black Trees**.
- Industry talks: 18 & 25 September.
- How are you doing? Is everything okay?
  - If you consider yourself at risk, please reach out!

Australian National University

COMP2100/6442
Software Design Methodologies / Software Construction

Parsing: Tokeniser & Parser

Bernardo Pereira Nunes and
Sergio J. Rodríguez Méndez

Outline

> Motivation: How are source codes analysed, compiled, and executed?

> Structured text

> Tokenisation

> Parsing

> Grammars

Unstructured Information

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Download as PDF
Printable version

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]

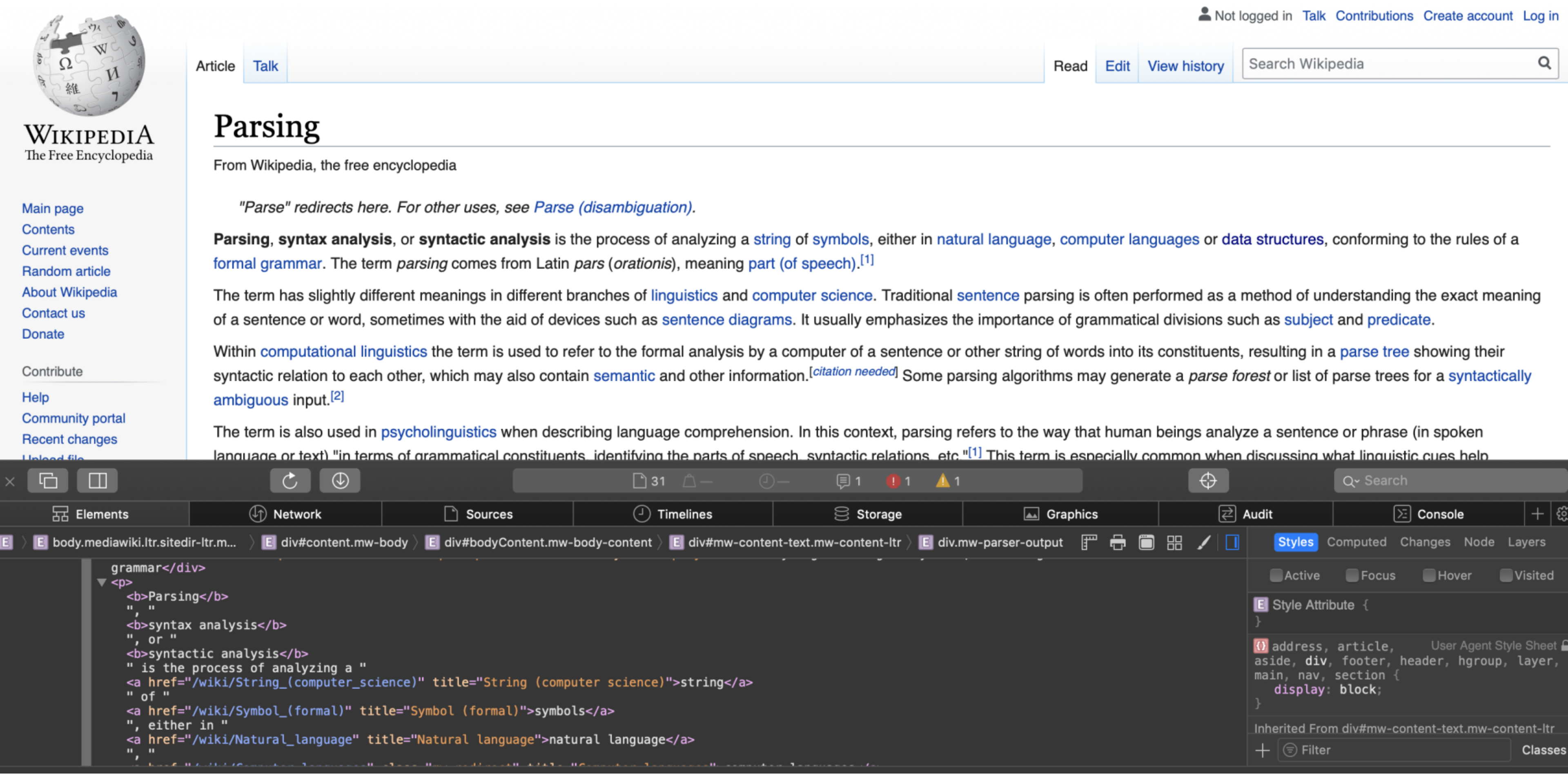
The term is also used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings analyze a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc."[1]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

Within computer science, the term is used in the analysis of computer languages, referring to the syntactic analysis of the input code into its component parts in order to facilitate the writing of compilers and interpreters. The term may also be used to describe a split or separation.

Contents [hide]

Isn't it funny that the Parsing article on Wikipedia is not structured?

Unstructured Information



Unstructured Information

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Wikidata item
Cite this page

In other projects
Wikimedia Commons
Wikispecies

Print/export

Redback spider

From Wikipedia, the free encyclopedia

The **redback spider** (***Latrodectus hasselti***), also known as the **Australian black widow**,[2][3][4] is a species of highly venomous spider believed to originate in South Australia or adjacent Western Australian deserts, but now found throughout Australia, Southeast Asia and New Zealand, with colonies elsewhere outside Australia.[5] It is a member of the cosmopolitan genus *Latrodectus*, the widow spiders. The adult female is easily recognised by her spherical black body with a prominent red stripe on the upper side of her abdomen and an hourglass-shaped red/orange streak on the underside. Females usually have a body length of about 10 millimetres (0.4 in), while the male is much smaller, being only 3–4 mm (0.12–0.16 in) long.

Mainly nocturnal, the female redback lives in an untidy web in a warm sheltered location, commonly near or inside human residences. It preys on insects, spiders and small vertebrates that become ensnared in its web. It kills its prey by injecting a complex venom through its two fangs when it bites, before wrapping them in silk and sucking out the liquefied insides. Often, it first squirts its victim with what resembles 'superglue' from its spinnerets, immobilising the prey by sticking the victim's limbs and appendages to its own body. The redback spider then trusses the victim with silk. Once its prey is restrained, it is bitten repeatedly on the head, body and leg segments and is then hauled back to the redback spider's retreat. Sometimes a potentially dangerous victim can be left to struggle for hours until it is exhausted enough to approach safely.[5] Male spiders and spiderlings often live on the periphery of the female spiders' web and steal leftovers. Other species of spider and parasitoid wasps prey on this species. The redback is one of few arachnids that usually display sexual cannibalism while mating.

After mating sperm is stored in the spermathecae, organs of the female reproductive tract, and can be used up to two years later to fertilise several clutches of eggs. Each clutch averages 250 eggs and is housed in a round white silken egg sac. The redback spider has a widespread distribution in Australia, and inadvertent introductions have led to established colonies in New Zealand, the United Arab Emirates, Japan and greenhouses in Belgium.[6]

The redback is one of the few spider species that can be seriously harmful to humans, and its liking for habitats in built structures has led it to being responsible for a large number of serious spider bites in Australia. Predominantly neurotoxic to vertebrates, the venom gives rise to the syndrome of latrodectism in humans; this starts with pain around the bite site, which typically becomes severe and progresses up the bitten limb and persists for over 24 hours. Sweating in localised patches of skin occasionally occurs and is highly indicative of latrodectism. Generalised symptoms of nausea, vomiting, headache, and agitation may also occur and indicate severe envenomation. An antivenom has been available since 1956.

Contents [hide]

Redback spider

Adult red back spider

Adult male (considerably smaller than female)

Scientific classification

Kingdom:	Animalia
Phylum:	Arthropoda
Subphylum:	Chelicerata
Class:	Arachnida
Order:	Araneae

Structured Information

DBpedia   Browse using   Formats   Faceted Browser   Sparql Endpoint

About: Redback spider

An Entity of Type : Spinnentier, from Named Graph : http://dbpedia.org, within Data Space : dbpedia.org

العنكبوت الأحمر الظهر (الاسم العلمي: Latrodectus hasseltii)، عنكبوت سام يعيش في أستراليا تتميز هذا العناكب بالشريط الأحمر في ظهرها.

Property	Value
dbo:abstract	▪ The redback spider (Latrodectus hasseltii) is a species of venomous spider indigenous to Australia. It is a member of the cosmopolitan genus Latrodectus, the widow spiders. The adult female is easily recognised by her spherical black body with a prominent red stripe on the upper side of her abdomen and an hourglass-shaped red/orange streak on the underside. Females have a body length of about 10 millimetres (0.4 in), while the male is much smaller, being only 3–4 mm (0.12–0.16 in) long. Mainly nocturnal, the female redback lives in an untidy web in a warm sheltered location, commonly near or inside human residences. It preys on insects, spiders and small vertebrates that become ensnared in its web. It kills its prey by injecting a complex venom through its two fangs when it bites, before wrapping them in silk and sucking out the liquefied insides. Male spiders and spiderlings often live on the periphery of the female spiders' web and steal leftovers. Other species of spider and parasitoid wasps prey on this species. The redback is one of few arachnids which usually display sexual cannibalism while mating. The sperm is then stored in the spermathecae, organs of the female reproductive tract, and can be used up to two years later to fertilise several clutches of eggs. Each clutch averages 250 eggs and is housed in a round white silken egg sac. The redback spider has a widespread distribution in Australia, and inadvertent introductions have led to established colonies in New Zealand, Japan, and in greenhouses in Belgium. The redback is one of the few spider species that can be seriously harmful to humans, and its preferred habitat has led it to being responsible for the large majority of serious spider bites in Australia. Predominantly neurotoxic to vertebrates, the venom gives rise to the syndrome of latrodectism in humans; this starts with pain around the bite site, which typically becomes severe and progresses up the bitten limb and persists for over 24 hours. Sweating in localised patches of skin occasionally occurs and is highly indicative of latrodectism. Generalised symptoms of nausea, vomiting, headache, and agitation may also occur and indicate severe poisoning. An antivenom has been available since 1956, and there have been no deaths directly due to redback bites since its introduction. (en)
dbo:thumbnail	▪ wiki-commons:Special:FilePath/Latrodectus_hasseltii_close.jpg?width=300
dbo:wikiPageExternalLink	▪ http://bie.ala.org.au/species/Latrodectus+hasseltii

http://dbpedia.org/page/Redback_spider

Structured Text

> Information is often stored in unstructured text files

> We often input information to computer via text files

>> Markup language: HTML, JSon, XML, Markdown…

>> Programming language: 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

Parsing

> Also called syntax (syntactic) analysis

> Aim to **understand** the exact meaning of **structured text**

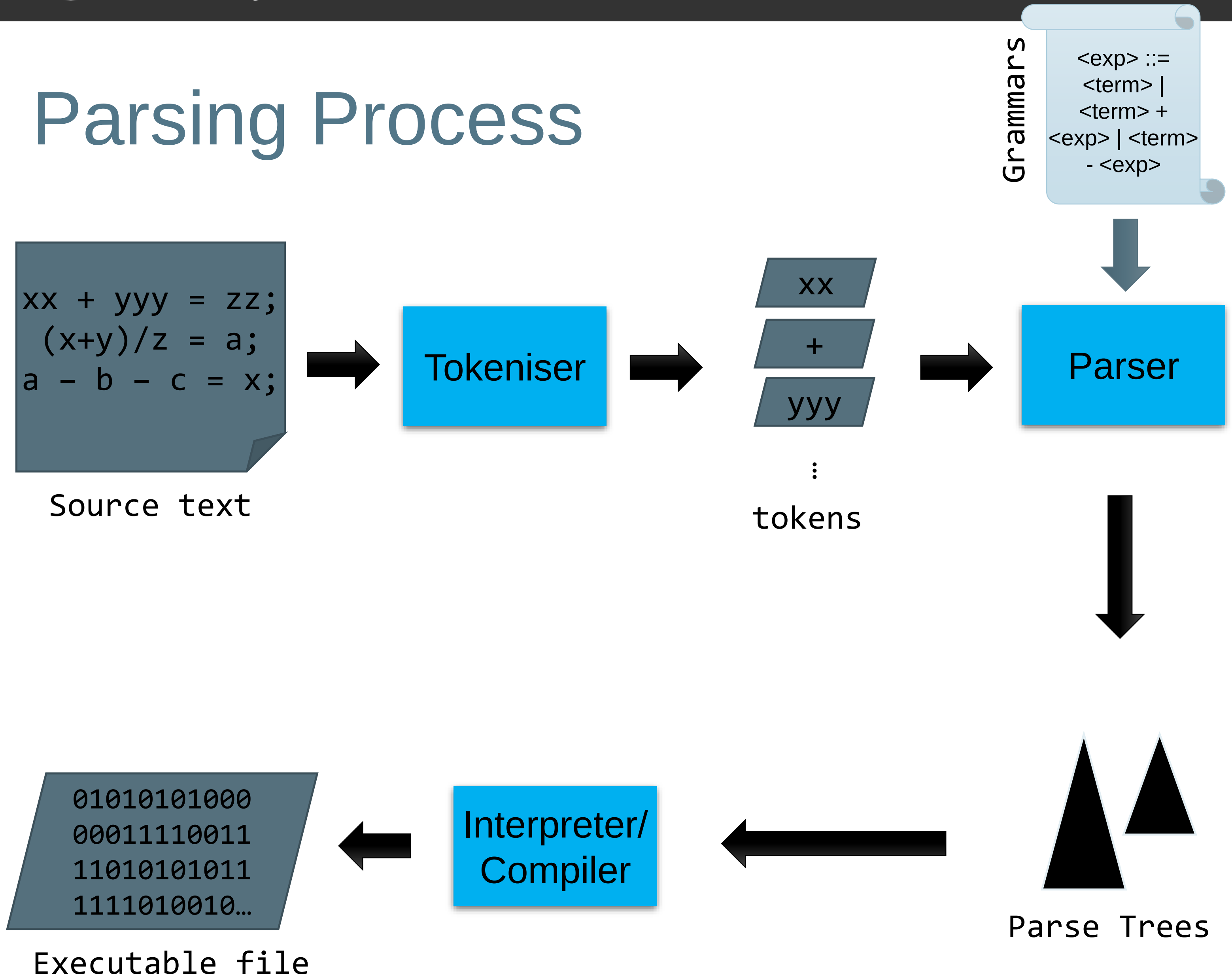
>> Resulting in parse tree, **a representation of structured text**

>> Preceded by a **tokeniser**

> **Need a grammar** to generate parse tree

>> Test whether a text conforms to the grammar

Parsing Process



Tokenisation

From text to tokens

aka lexical analyser

Tokenisation

…*is the process of converting a sequence of characters into a sequence of tokens!*[1]

Natural language tokenisation:

"I want to tokenise this sentence."

→ I / want / to / tokenise / this / sentence / .

In this case, each token is a vocabulary word or punctuation marks

Some languages may not be as easy as this one (e.g., ***scriptio continua***)!

[1] https://en.wikipedia.org/wiki/Lexical_analysis

Tokenisation in Structured Text

> A token is a **string with an assigned meaning** structured as a pair consisting of a token kind (type) and an optional token value.

Example: `public void myMethod(int var1)`

(**keyword**, “public”)

(**keyword**, “void”) (**identifier**, “myMethod”) (**keyword**, “int”) (**identifier**, “var1”)

(**punctuator**, “(“)

→ ((**keyword**, “public”), (**keyword**, “void”), (**identifier**, “myMethod”), (**punctuator**, “(“), (**keyword**, “int”), (**identifier**, “var1”) …)

sometimes with location information!

The Output

> A series of tokens: type, location, name (if any)
>> 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
>>> Floating point: 5.23e+2

Punctuators (Separators)

> Typically individual special characters such as ({ } : . ;

>> Sometimes double characters: tokeniser looks for longest token:

/*, //, -- comment openers in various languages

>> Returned as identity (type) of token

… and perhaps location for error messages and debugging purposes

Operators

> Like punctuators

>> No real difference for tokeniser

>>> Tokenisers do not “process/execute” tokens

>> Typically single or double special chars

>>> Operators + - == <=

>>> Operations :=

>> Returned as type of token

… and perhaps location

Identifier & Keywords

> **Identifier**: function names, variable names

>> Length, allowed characters, separators

> Need to build a names table

>> Single entry for all occurrences of var1, myFunction

>> Typical structure: hash table

> Tokeniser returns token type

… and 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

> 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., Java, Javascript (https://javascript-minifier.com), …

> Fixed format languages (historical)

>> Layout is critical

>>> Fortran

>>> Python, indentation

>> Tokeniser must know about layout to find tokens

>> It was born in 1950's (punched cards, one statement per card – easy to debug/maintain code)

Pros and Cons?

Fixed is readable and easy to understand; Free is flexible and can produce smaller file sizes.

Case Equivalence

> Some languages are **case-insensitive**
  >> Pascal, Ada
> Some are **case-sensitive**
  >> C, Java
> Tokeniser ignores case if needed
  >> This_Routine == THIS_RouTine
  >> Error analysis may need exact casing

General Approach

> Define set of token kinds:

An enumeration type (tok_int, tok_if, tok_plus, tok_left_paren, etc)

> Tokeniser 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 Tokeniser

```
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();

}
```

Example code for this lecture is in our REPO.

```java
public class MySimpleTokenizer extends Tokenizer {
       private String text;           // save text
       private int pos;               // current position
       private Object current;        // save token extracted
       ...
       static final char symbol[] = {'*', '+', '(', ')', ';'};

       public void next() {
              consumewhite(); // ignores all white spaces
              if (pos == text.length()) {
                     current = null; // end of text
              } else if (isin(text.charAt(pos), symbol)) {
                     current = "" + text.charAt(pos);
                     pos++;        //extract predefined symbol
              }
```

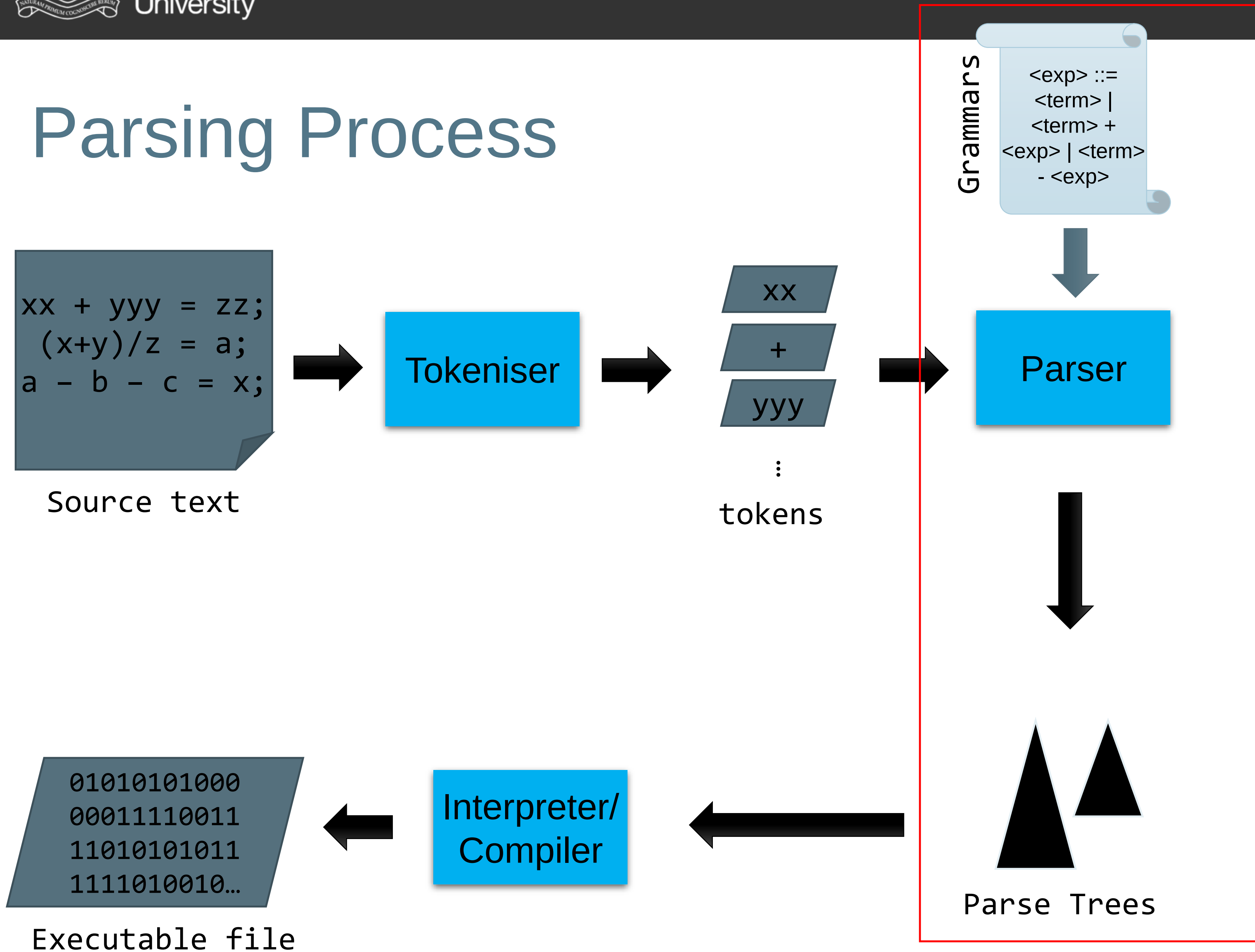
Parsing numeral

```
else if (Character.isDigit(text.charAt(pos))) {
        int start = pos;
        while (pos < text.length() &&
                Character.isDigit(text.charAt(pos)) )
                pos++;
        // Check period in a sequence. Note that valid double has only
single period in a sequence.
        if (pos+1 < text.length() && text.charAt(pos) == '.' &&
                Character.isDigit(text.charAt(pos+1))) {
                pos++;
                while (pos < text.length() &&
                        Character.isDigit(text.charAt(pos)))
                        pos++;
                current = Double.parseDouble(text.substring(start,
                        pos));
        } else {
                current = Integer.parseInt(text.substring(start,
                        pos));
        }
}
```

Parsing

From tokens to tree

Parsing Process



Grammars

> **Grammars** express languages

>> It specifies the rules of a language

Example

English grammar (simplified)

```
<sentence> → <noun_phrase> <predicate>
<noun_phrase> → <article> <noun>
<predicate>  → <verb>
```

a → *b* means that *a* can be rewritten as *b*.

```
<article> → a
<article> → the

<noun> → cat
<noun> → dog

<verb> → runs
<verb> → sleeps
```

Derivation of string “the dog sleeps”:

```
<sentence> ⇒ <noun_phrase> <predicate>
           ⇒ <noun_phrase> <verb>
           ⇒ <article> <noun> <verb>
           ⇒ the <noun> <verb>
           ⇒ the dog <verb>
           ⇒ the dog sleeps
```

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

Derivation of string "a cat runs":

```
<sentence> ⇒ <noun_phrase> <predicate>
           ⇒ <noun_phrase> <verb>
           ⇒ <article> <noun> <verb>
           ⇒ a <noun> <verb>
           ⇒ a cat <verb>
           ⇒ a cat runs
```

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

Language of the grammar

Language: all possible derivations

```
L = {   “a cat runs”,
        “a cat sleeps”,
        “the cat runs”,
        “the cat sleeps”,
        “a dog runs”,
        “a dog sleeps”,
        “the dog runs”,
        “the dog sleeps” }
```

Context-Free Grammars (CFG)

> Grammars provide a precise way of specifying language

> A context-free grammar is often used to define the syntax

> **A context-free grammar is specified via a set of production rules** (→ or ::=) **with**

>> Variables (or non-terminals; surrounded with <>)

>> Terminals (symbols)

>> Alternatives (|)

https://en.wikipedia.org/wiki/Context-free_grammar

Productions / Variable and Terminals

Variables

$$\langle \mathit{noun} \rangle \to \overbrace{\mathit{cat}}^{\text{Terminal (symbol)}}$$

$$\langle \mathit{sentence} \rangle \to \underbrace{\langle \mathit{noun\_phrase} \rangle \ \langle \mathit{predicate} \rangle}_{\text{Sequence of Variables}}$$

Alternatives |

Conventional notation

$$\langle article \rangle \to a$$
$$\langle article \rangle \to the$$

$$\Rightarrow$$

$$\langle article \rangle \to a \mid the$$

Example – Grammar for Integer (e.g. 25)

$$< num > \rightarrow < digit >< num > | < digit >$$

$$< digit > \rightarrow 0|1|2|3|4|5|6|7|8|9$$

It includes numbers such as 0111!

Question: how to have an unsigned integer number not starting from zero?

https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/

Example

Grammar for integer

Question: how to have an unsigned integer number not starting from zero?

S → 0 | N

D → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A → 0 | D

F → AF | A

N → DF | D

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

https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/

Formal Definitions

Grammar: $G = (V, T, S, P)$

Set of variables

Set of terminal symbols

Start variable

Set of productions

With example

productions

$$< num > \rightarrow < digit >< num > \mid < digit >$$
$$< digit > \rightarrow 0|1|2|3|4|5|6|7|8|9$$

$$G = (V, T, S, P)$$

$$V = \{< num >, < digit >\}$$

variables

$$T = \{0, 1, 2, \ldots, 9\}$$

terminals

$$S = < num >$$

start variable

Grammar for mathematical expressions

P = 4 production rules

$$\boxed{E \rightarrow E + E \ \mid E * E \mid (E) \mid a}$$

$V = \{E\}$

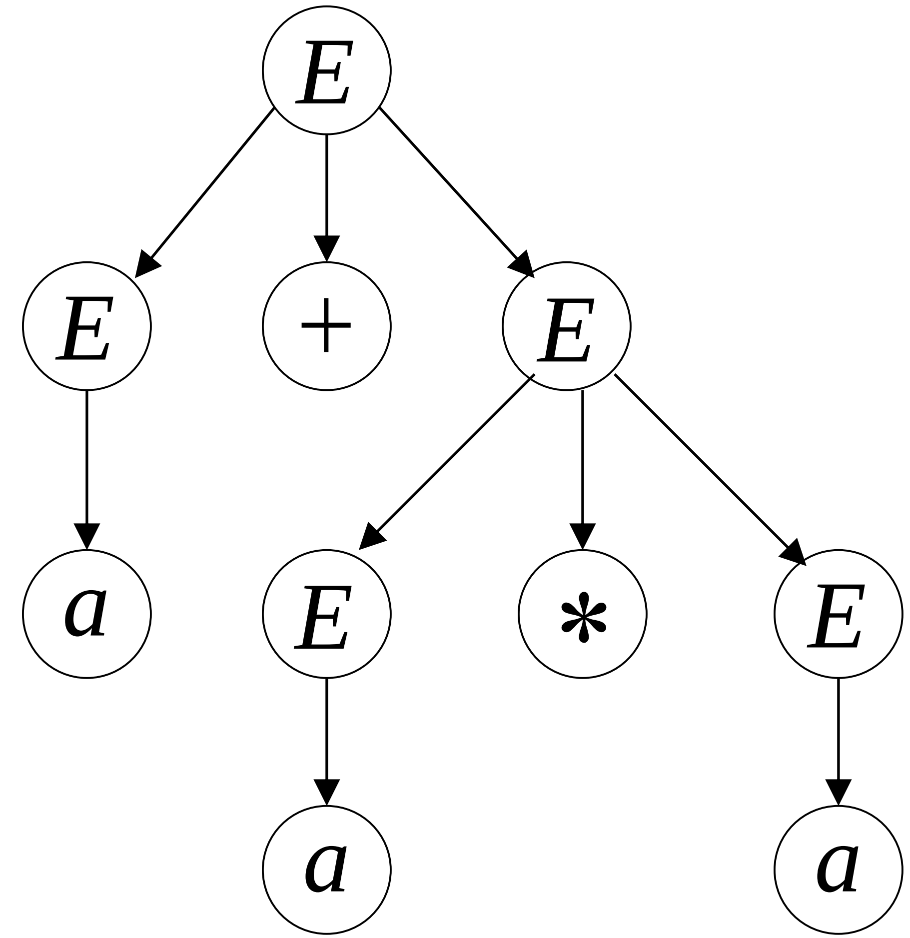
$T = \{a, +,\ *, (,\ )\}$

Let's assume 'a' can be any number

$S = E$

All mathematical expression starts from E

Derivation for $a + a * a$

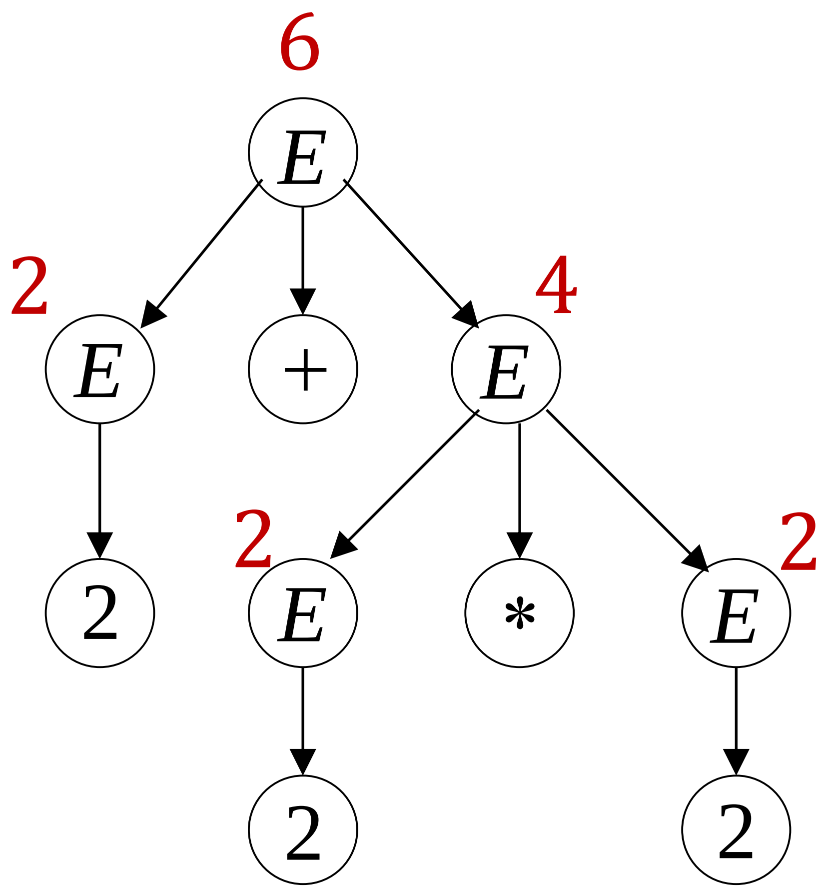


$$E \rightarrow E + E \mid E * E \mid (E) \mid a$$

$$\begin{aligned} E &\Longrightarrow E + E \\ &\Longrightarrow a + E \\ &\Longrightarrow a + E * E \\ &\Longrightarrow a + a * E \\ &\Longrightarrow a + a * a \end{aligned}$$

Parse Tree!

Evaluate expression via Tree

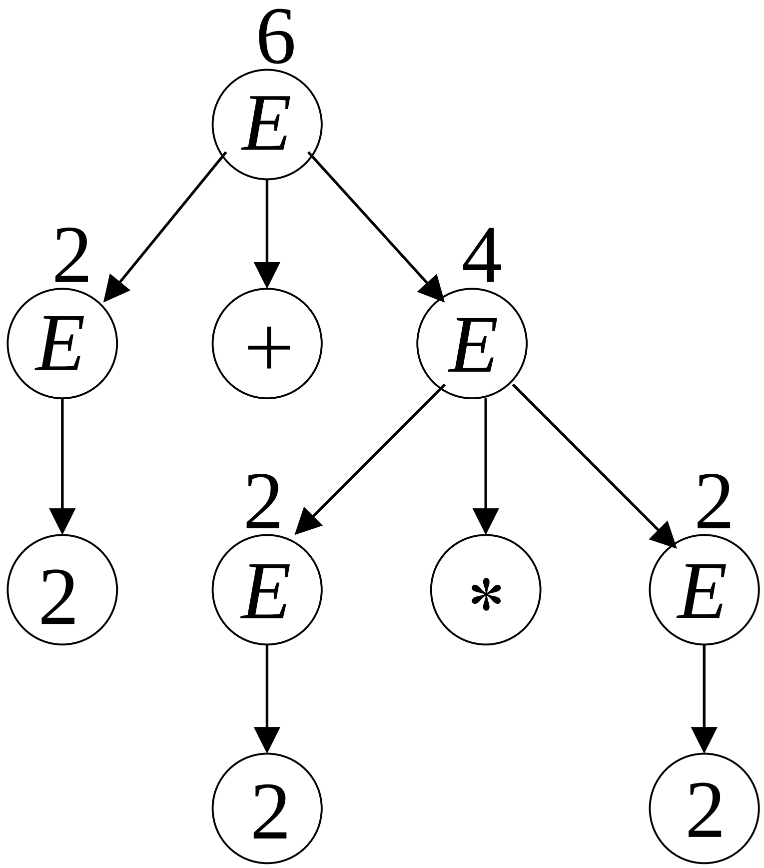


> Parsing tree defines an evaluation order

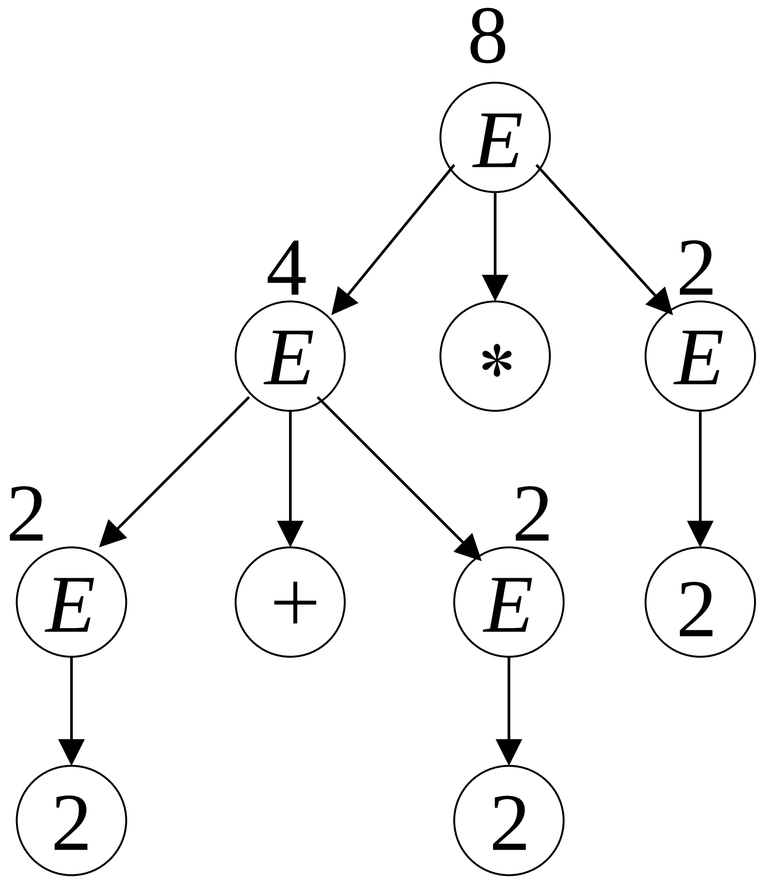
> Evaluation from leaves to root

> Intermediate variables are replaced by terminal values

Ambiguity! It is ambiguous because there are two possible derivations!



Parse tree 1



Parse tree 2

Ambiguous Grammar

A context-free grammar G is ambiguous

> if there is a string *w* from the language of grammar G which has

>> more than one derivation tree

Example

$$\boxed{E \to E + E \ \mid E * E \mid (E) \mid a}$$

Remove ambiguity

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E) | a$$

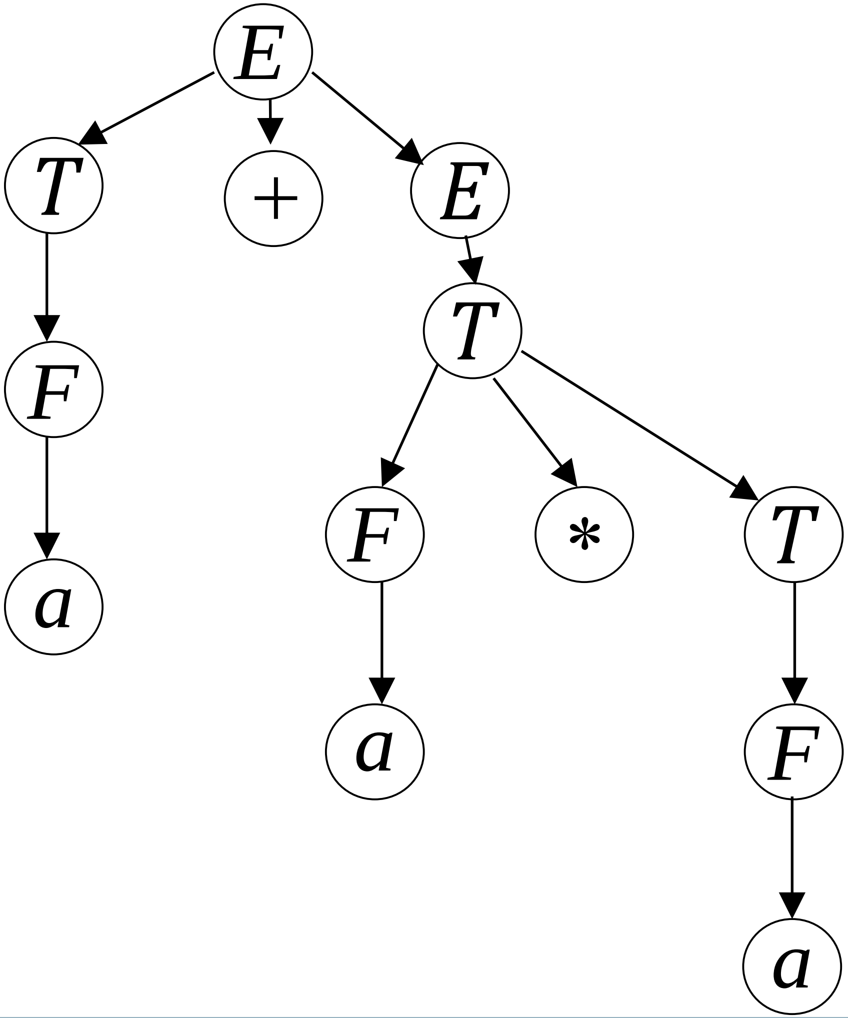
Ambiguous Grammar

$$E \rightarrow T + E | T$$
$$T \rightarrow F * T | F$$
$$F \rightarrow (E) | a$$

Non-ambiguous Grammar

Unique derivation tree for $a + a \, * a$

$$E \rightarrow T + E | T$$
$$T \rightarrow F * T | F$$
$$F \rightarrow (E) | a$$



Unique derivation tree for $a + a \ast a$

Using the CFG Developer – Stanford:

S → T+S | T

T → F*T | F

F → (S) | a

a → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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.

2+2*2

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 a parser

Recursive-descent parser

> Top-down parser

> **Parse from the start variable,** recursively parse input tokens

> Create a method for each left-hand side variable in the grammar

> These methods are responsible for generating parsed nodes

Variable (& Symbol) as a node

To construct a parse tree,

> we can adopt ideas from binary search tree

> each variable (& symbol) can be represented as a node in a tree

> Define a node class for each variable

- Given grammar start with Exp:

  Exp → Term + Exp | Term

  Term → Factor * Term | Factor

  Factor → (Exp) | a

- Node classes
  - public abstract class Exp
  - public class Term extends Exp
  - public class Factor extends Exp
  - public class Int extends Exp
  - public class AddExp extends Exp
  - public class MulExp extends Exp

Implement Parser

Given grammar:

Exp → Term + Exp | Term

Term → Factor * Term | Factor

Factor → (Exp) | a

We can define a parse method for each variable

To get a next token (You may set tokeniser as a member of parser class)

```
public Exp parseExp(Tokenizer)
```

Return type: Exp node (abstract class)

Pseudo code for parsing

Exp → Term + Exp | Term

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

Both production rule starts with Term

If the next token is +, apply first production rule

Addition between term and exp

Pseudo code for parsing
Term → Factor * Term | Factor

Both production rule starts with Factor

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

If the next token is *, apply first production rule

Multiplication between Factor and Term

Australian National University

Pseudo code for parsing Factor → (Exp) | a

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

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

Remove ‘)’

The second production rule

RDP - Limitations

The Recursive Descent Parsing approach will not work with left recursive grammars. For example:

$$< \text{binary} > \rightarrow < \text{binary} >< \text{digit} > | < \text{digit} >$$

$$< \text{digit} > \rightarrow 0|1$$

> could not be parsed using the recursive descent parser

However, we could transform the grammar into:

$$< \text{binary} > \rightarrow < \text{digit} >< \text{binary} > | < \text{digit} >$$

$$< \text{digit} > \rightarrow 0|1$$

> which represents the same language, yet can be parsed by the recursive descent parser

Exercise

Which alternative is INCORRECT about tokenisation:
Select one:

a) The tokenisation process ends after identifying all tokens of a text and evaluating the expressions found. For instance, it is part of the tokenisation to detect the tokens of the expression "((1+2) * 3)" and show its result, in this case, 9.

b) A tokenisation process can be case-insensitive or case-sensitive since it is usual to find texts with lower and upper case words.

c) A token is defined as a string with an assigned meaning. It is structured as a pair consisting of a type and value. For instance, in the sentence "1, 2, 3", number and comma can be types of tokens.

d) Tokenisation is the process of converting a sequence of characters into a sequence of tokens. Considering the sentence "I am 30 years old.", the words "I", "years", "am", "old", the punctuation mark "." and the number "30" are examples of tokens.

Exercise

```java
public abstract class Tokenizer {
   public abstract void next();
   public abstract Token current();
   public abstract boolean hasNext();
}
```

Considering the following abstract class Tokenizer, which alternative is INCORRECT:

a) An implementation of the method hasNext() will be responsible for extracting the next token from a text and returning the token to the code that invoked it.

b) The method next() is responsible for finding the next current token. The current token will be returned by the method current() as an instance of the class Token.

c) The class Token is usually composed of a pair of type and value which is the definition of token. A good strategy is to use an enum as a token type.

d) The Tokenizer abstract class has no implementation. To perform a tokenisation process, you need to create a subclass of Tokenizer and then implement the abstract methods. Each subclass of Tokenizer can have its own implementation of each method.

Exercise

Considering the grammar below and assuming that there is a parser for it, which instruction is NOT acceptable for this grammar?

```
<command> := find <fields> in <table_name> having id = <id_value>
   <fields>     := <field>, <fields> | <field>
   <table_name> := <string_value>
   <id_value>   := <int_value>
   <field>      := <string_value>
```

Select one:

a) find name in animal having id = 1
b) find name in Employee having id = 40
c) find name; age; weight in Animal having id = 2
d) find street, number in Address having id = 1

Exercise

Which one of the following is acceptable by a language specified by the grammar:

S := ()

S :=)S(

S := SS

Select one:

a)))((

b) ()()))(

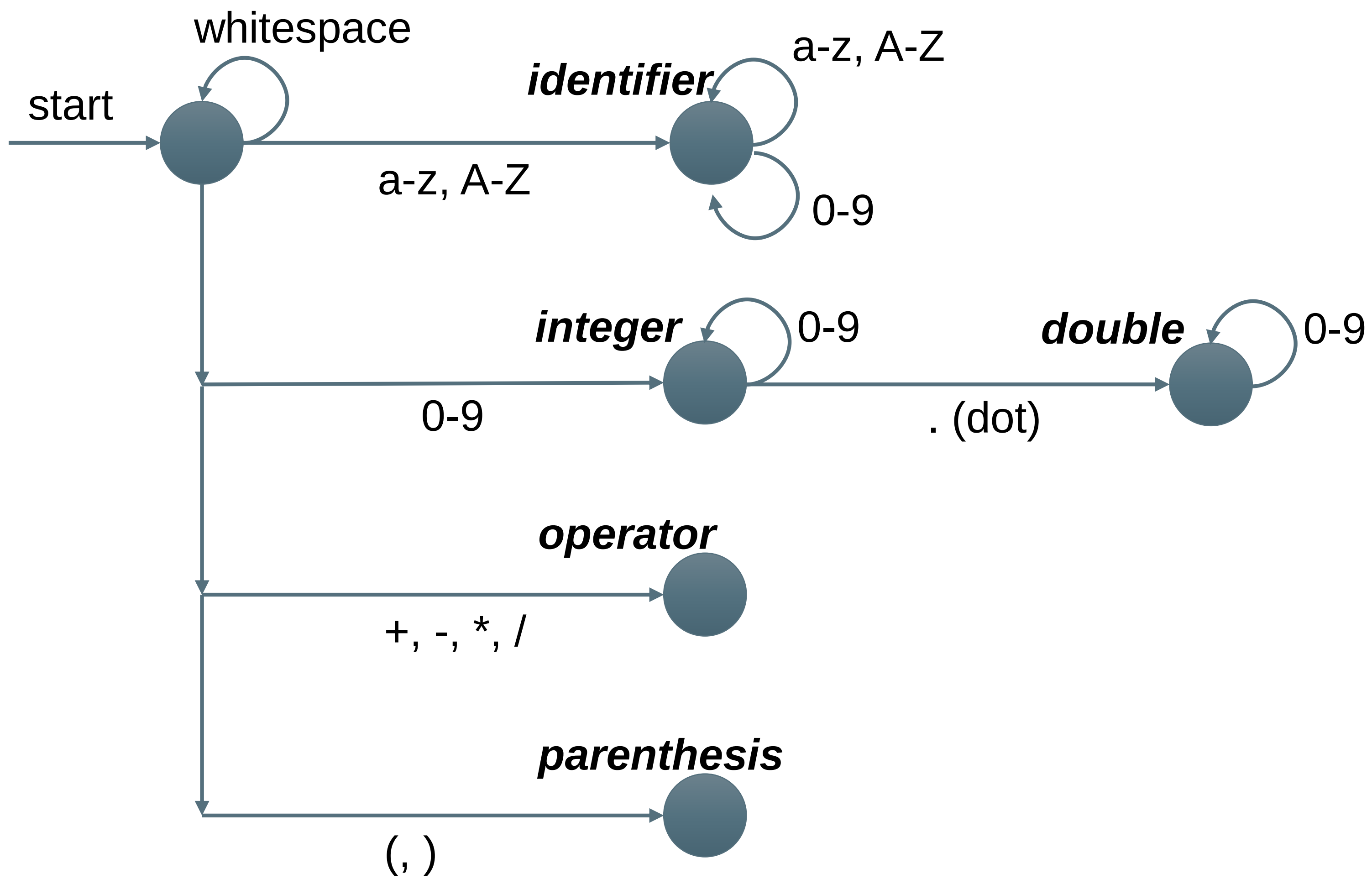
c))()()(

d))(()))(

Finite State Machine

- Tokeniser can be formalised via finite state machine (FSM)
- FSM consists of a set of
  - States (including initial and final state)
  - Transition between states
- FSM starts from the initial state
  - Move to the other state based on the next character
- FSM are the theoretical artefacts for RegExps.

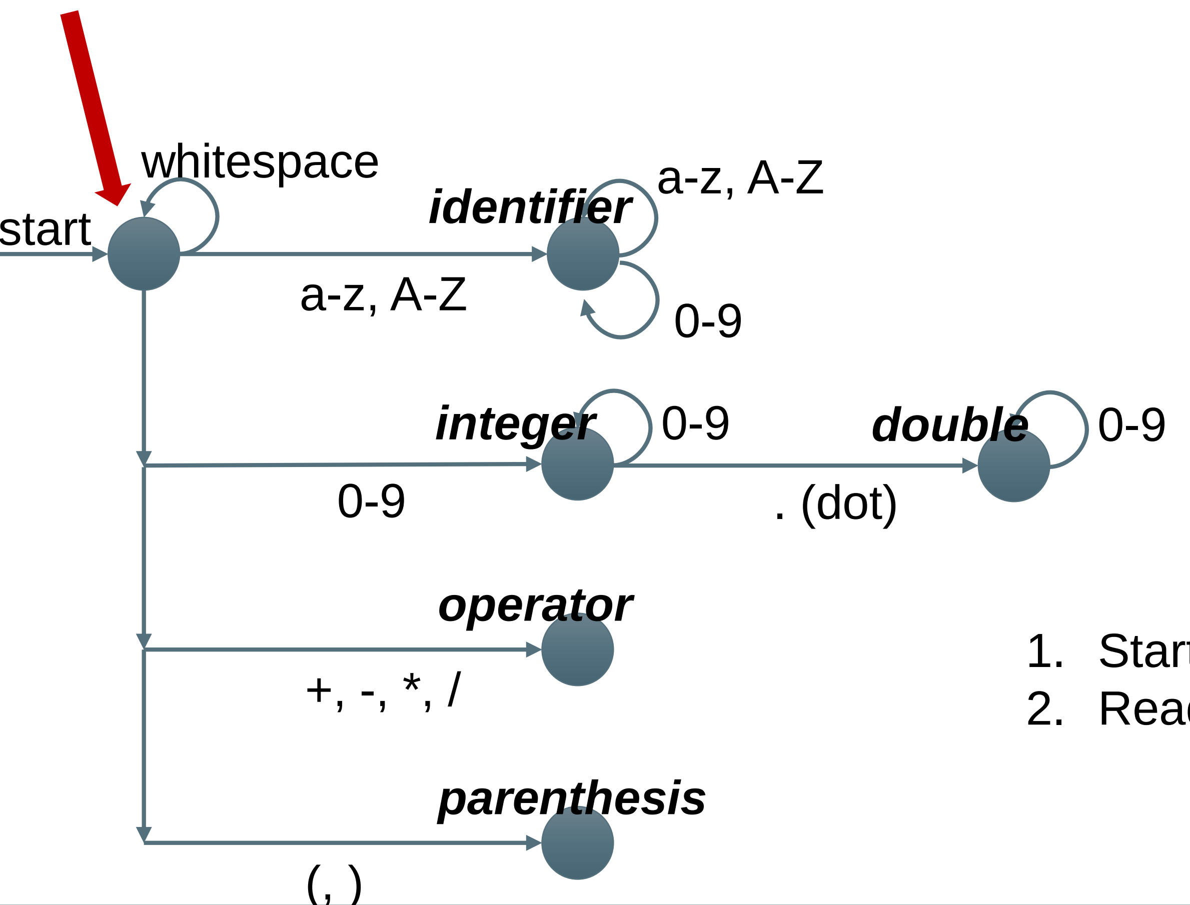
(Simplified) FSM for Tokeniser



: current state

: current position

Tokenising:
12.45+6

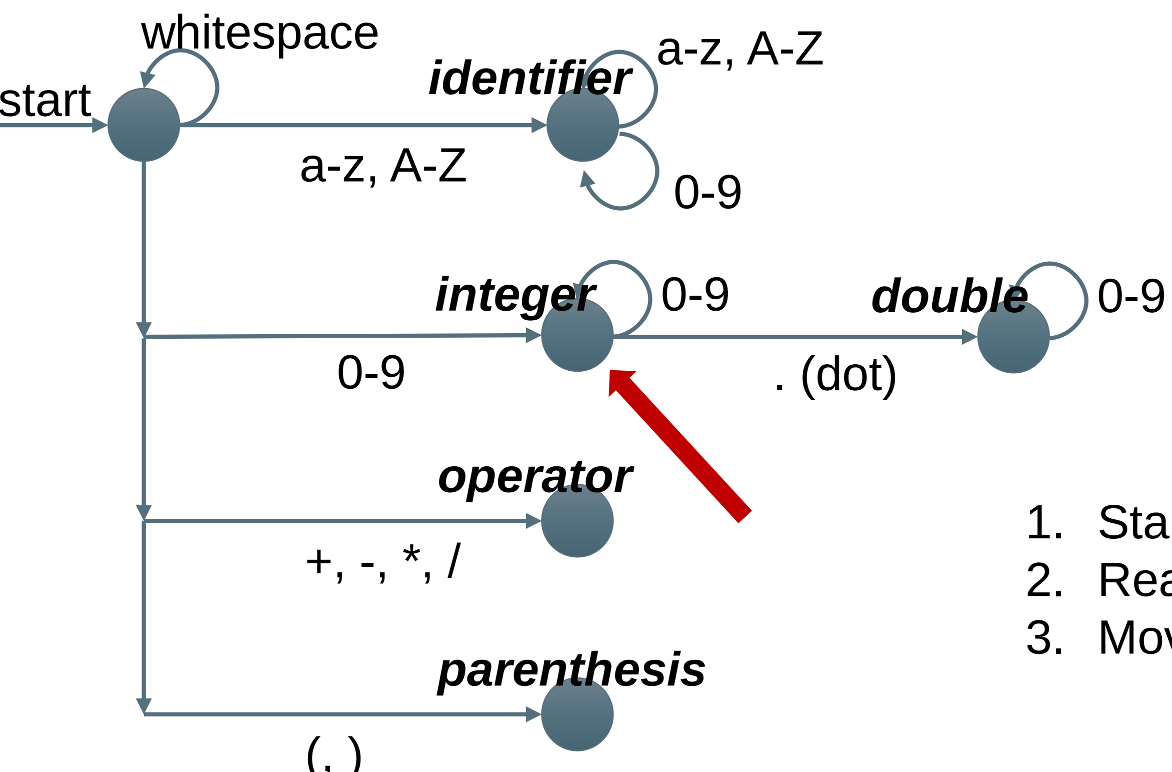


1. Starts from initial state
2. Read character from current position

: current state

: current position

Tokenising: 12.45+6

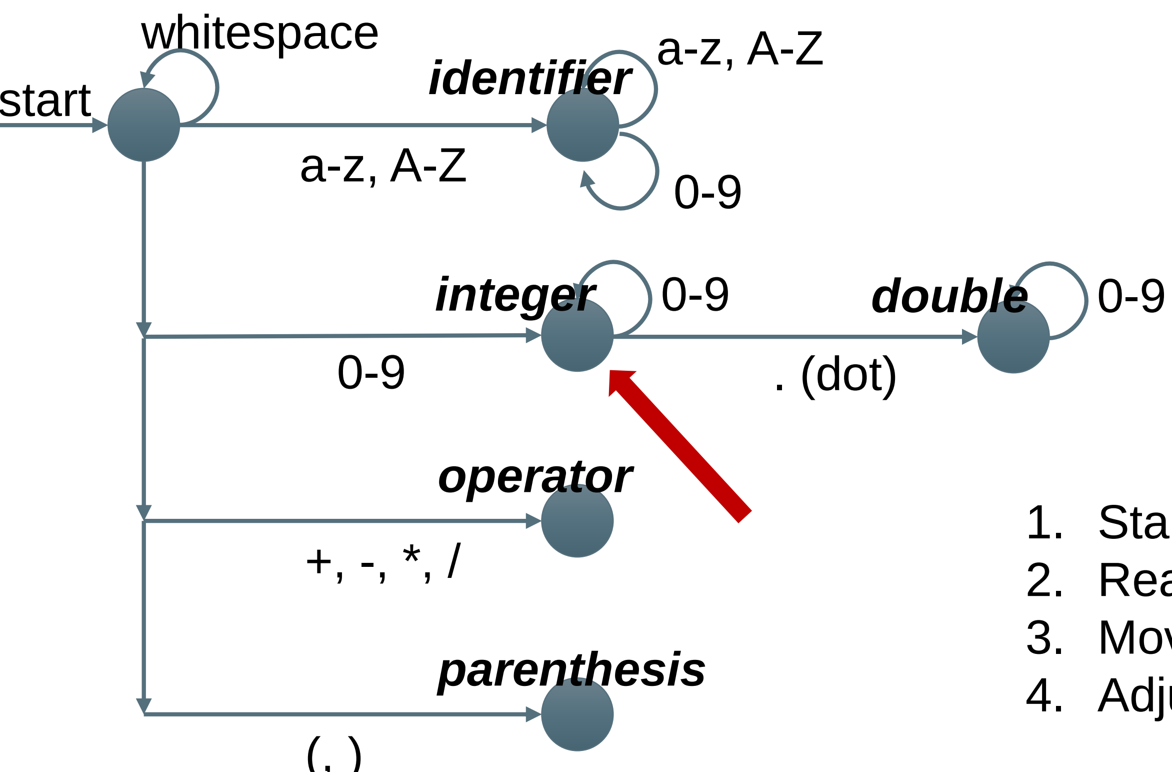


1. Starts from initial state
2. Read character from current position
3. Move state based on the read

: current state

: current position

Tokenising: 12.45+6

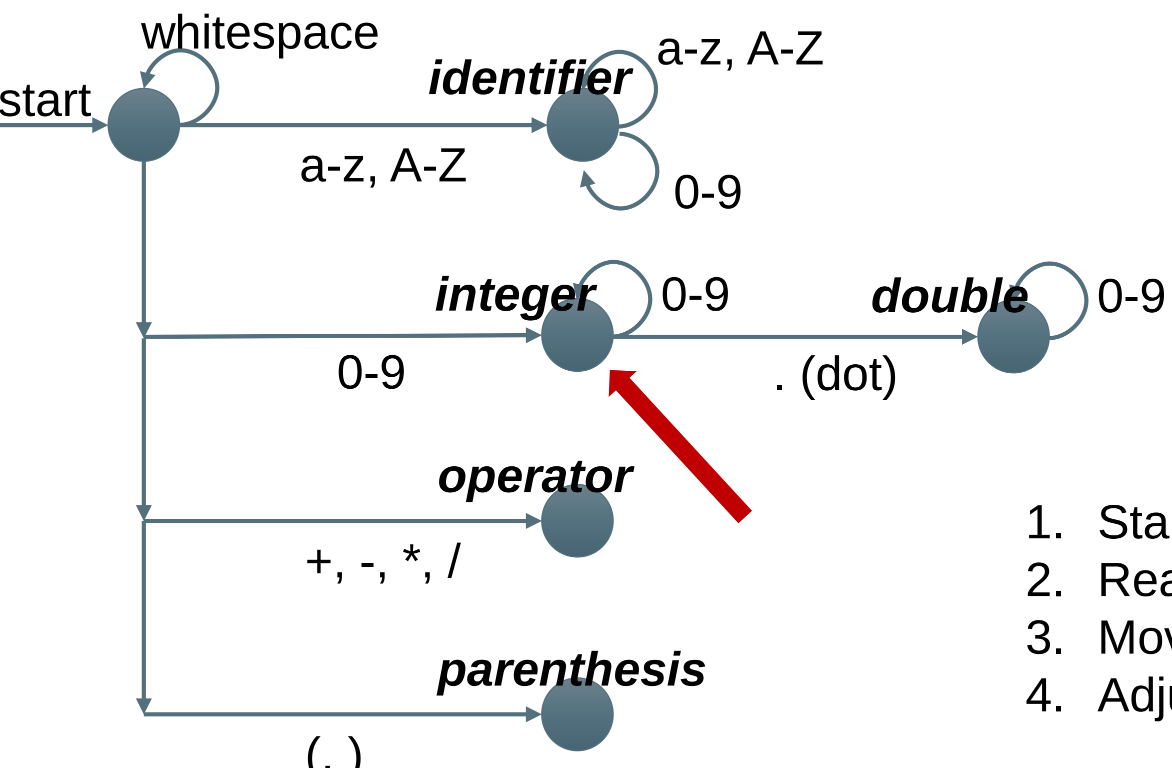


1. Starts from initial state
2. Read character from current position
3. Move state based on the read
4. Adjust position of tokeniser

: current state

: current position

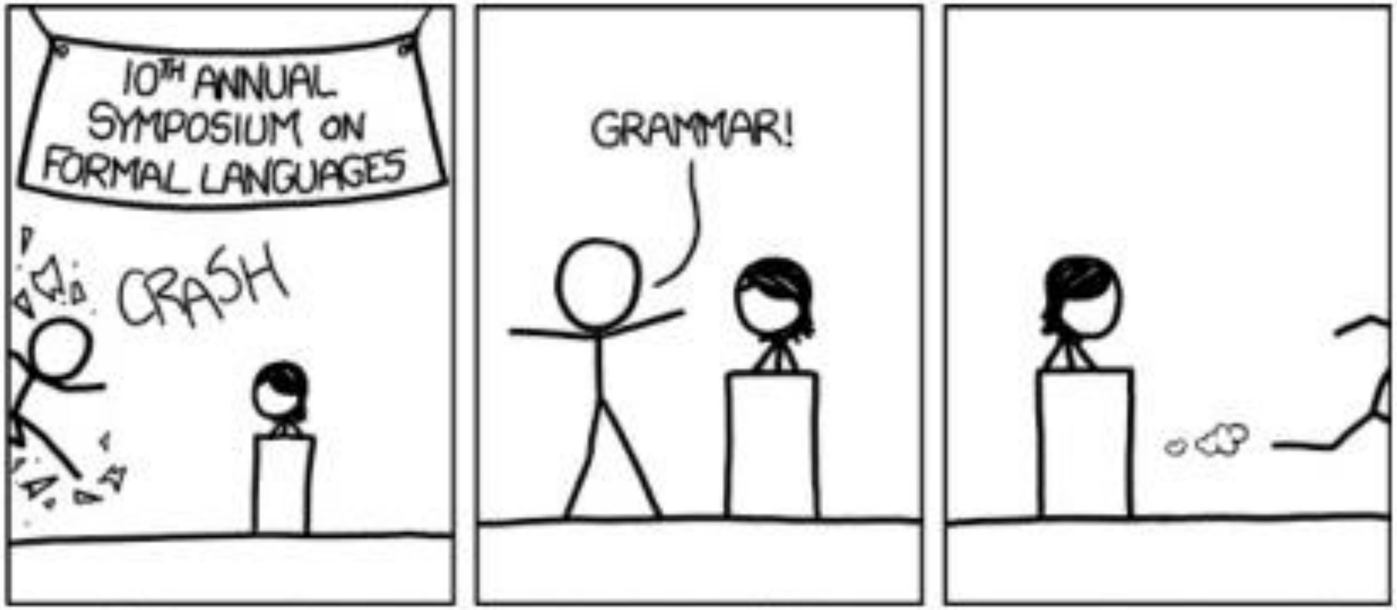
Tokenising: 12.45+6



1. Starts from initial state
2. Read character from current position
3. Move state based on the read
4. Adjust position of tokeniser

Meme for today's lecture! Keep practicing!

Formal Languages



Title text: [audience looks around] 'What just happened?' 'There must be some context we're missing.'

https://www.xkcd.com/1090/

Well… context-free grammars…