Theme 1
Compilers, Languages and Grammars
Introduction

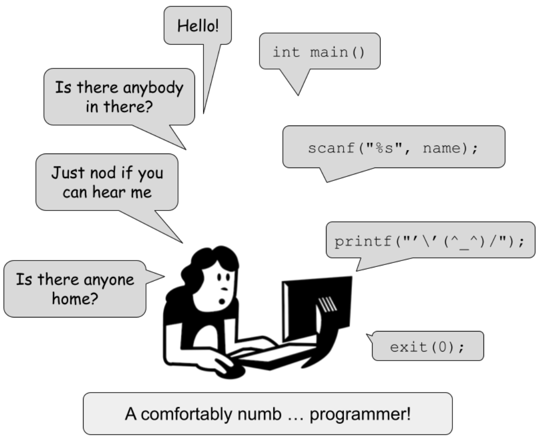
Compilers, 2nd semester 2024-2025

Miguel Oliveira e Silva, DETI, University of Aveiro

Contents

1 Frame

- In this course we will talk about *languages* – what they are and how we can define them – and about *compilers* – tools that recognize them and allow you to perform actions as a result of that process.


Hello!
int main()
Is there anybody in there?
scanf("%s", name);
Just nod if you can hear me
printf("'\'(^_^)/");
Is there anyone home?
exit(0);
A comfortably numb … programmer!

- If you had to define *language* how would you do it?
  - We can say that it is a protocol that allows us to *express, transmit and receive ideas*.
  - Until the middle of the last century, it would be said that it was a form of *communication* between people or, eventually, between living beings.
  - With the advent of computers, the concept of language was extended to communication with and between machines.
  - In common, the need to have more than one communicating entity, and a code and set of rules that makes this communication intelligible to all parties.
  - A person trying to communicate in Portuguese with someone who does not know that language is as effective as try to get a cat to play a sheet of a Beethoven sonata (another language: music).
- Therefore, it is necessary not only to have an encoding and a set of rules suitable for each language, as well as interlocutors who know them.
- Interestingly, different languages like Portuguese and English are composed of different *words*, but they share many of the symbols used to construct those words.
- Thus, "*goodbye*" and "*goodbye*" are different sequences of letters, even though they have a similar meaning and also share the alphabet of letters with which they are constructed.
- However, it hardly recognizes any sharing with the translation into one of the first written languages – the Babylonian cuneiform language ($\sim$ 3400BC)[1]:

  [illegible]

- On the other hand, there are also words that are the same with different meanings (depending on the context):
  - *hill*, *river*, *path*, ...
- On the other hand, our understanding of a text is not only done by reading the sequence of letters that compose it, but understanding its sequence of *ideas*, which are expressed by *phrases*, which in turn are sequences grammatically correct sequences of *words*, they are composed of sequences of *letters* and other alphabet symbols.

- Different languages may use different symbols (letters or characters), or share many of them.
- Comprehension of a word is done letter by letter, but this does not happen with a text.

[1]Translated at: `https://funtranslations.com/babylonian`

- Thus, we can see a natural language like Portuguese as being composed of more than one language:
  - One that spells out the rules for building words from the alphabet of letters:

$$\mathrm{a} + \mathrm{d} + \mathrm{e} + \mathrm{u} + \mathrm{s} \;\rightarrow\; \text{goodbye}$$

  - And another one that contains the grammar rules for building sentences from the resulting words from the previous language:

$$\text{goodbye} + \text{and} + \text{see you} + \text{tomorrow} \;\rightarrow\; \text{goodbye and see you tomorrow}$$

    In this case, the alphabet ceases to be the set of letters and becomes the set of existing words.
- Inherent in languages is the need to decide whether a sequence of alphabet symbols is valid.
  - **correct**:
    $\mathrm{a} + \mathrm{d} + \mathrm{e} + \mathrm{u} + \mathrm{s} \rightarrow \text{goodbye}$
    $\text{goodbye} + \text{and} + \text{until} + \text{tomorrow} \rightarrow \text{goodbye and see you tomorrow}$
  - **incorrect**:
    $\mathrm{e} + \mathrm{d} + \mathrm{u} + \mathrm{a} + \mathrm{s} \rightarrow \text{edes}$
    $\text{until} + \text{goodbye} + \text{tomorrow} + \text{and} \rightarrow \text{see you tomorrow and}$
- Only valid strings allow correct communication.
- On the other hand, this communication often has an effect.
- Whether that effect is a response to the initial message, or the triggering of any action.

1.1 Programming languages

- The languages for communicating with computers – called programming languages – share all these characteristics.
- They differ in that they cannot have any *ambiguity*, and that the triggered actions often be a change in the state of the computational system, which may be linked to entities such as other computers, people, robotic systems, washing machines, etc.
- Let's see that we can define programming languages by well-behaved formal structures.
- Furthermore, we will also see that these definitions help us to implement interesting actions.

Development of programming languages umbilically linked with compilation technologies!

2 Compilers: Introduction

Compilers: Comprehension, interpretation and/or automatic translation of languages.

- *compilers* are programs that allow:
  1. decide on the correction of sequences of symbols of the respective alphabet;
  2. trigger actions resulting from those decisions.
- Compilers are often "limited" to translating between languages.

$$\begin{tikzcd}\text{source language} \arrow[r] & \boxed{\texttt{Compiler}} \arrow[r] & \text{target language}\end{tikzcd}$$

- This is the case for compilers of high-level programming languages (`Java`, `C++`, `Eiffel`, etc.), which translate the source code of these languages in code of languages closer to the *hardware* of the system computational (e.g. *assembly* or *Java bytecode*).
- In these cases, in the absence of errors, a program composed of executable code is generated directly or indirectly by the computer system:

$$\begin{tikzcd}\text{input} \arrow[r] & \boxed{\texttt{Program}} \arrow[r] & \text{output}\end{tikzcd}$$

Example: Java *bytecode*

```
public class hello
{
   public static void main(String[] args)
   {
     System.out.println("Hello!");
   }
}
```

```
javac Hello.java
javap -c Hello.class
```

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2 // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3 // String Hello!
       5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

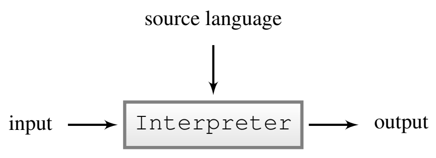
Example 2: Calculator

- Source code:

-
  ```
  1+2*3:4
  ```
- A possible compilation for Java:

```
public class CodeGen {
   public static void main(String[] args) {
      double v2 = 1;
      double v5 = 2;
      double v6 = 3;
      double v4 = v5 * v6;
      double v7 = 4;
      double v3 = v4 / v7;
      double v1 = v2 + v3;
      System.out.println(v1);
   }
}
```

- A possible variant consists of an *interpreter*:



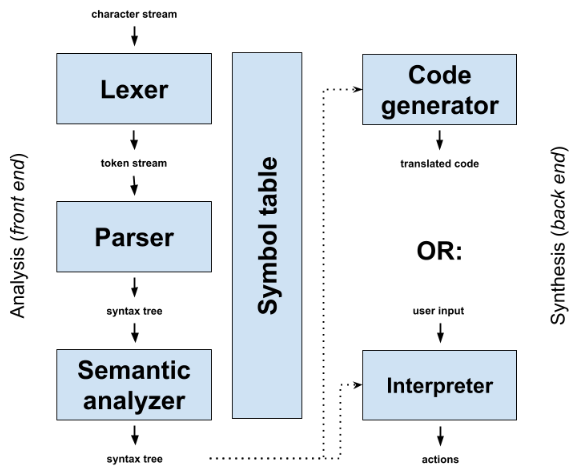
- In this case, execution is carried out instruction by instruction.
- `Python` and `bash` are examples of interpreted languages.
- There are also hybrid approaches where there is compilation of code for a language intermediate, which is then interpreted at runtime.
- The language `Java` uses a strategy like this where the source code is compiled to *Java bytecode*, which is then interpreted by the `Java` virtual machine.
- In general, compilers process source code in *text* format, with a wide variety in the format of the generated code (text, binary, interpreted, . . .).

Example: Calculator

- Source code:
-
```
1+2*3:4
```

- One possible interpretation:
```
2.5
```

3 Structure of a Compiler



- An interesting feature of high-level language compilation is the fact that that, as in the case of natural languages, this compilation involves more than one language:
  - **lexical analysis**: composition of letters and other characters into words (*tokens*);
  - **parsing**: composition of *tokens* into a suitable syntactic structure.
  - **semantic analysis**: checking – as much as possible – that the syntactic structure has meaning.
- Actions consist of generating the program in the target language and may involve also different stages of code generation and optimization.

3.1 Lexical Analysis

- Conversion of the input string into a string of lexical elements.
- This strategy greatly simplifies the grammar of parsing, and allows for an implementation very efficient lexical analyzer (later we will see in detail why).
- Each lexical element can be defined by a tuple with an element id and its value (value can be omitted when not applicable):

  ```
  <token_name , attribute_value >
  ```

- Example 1:

  ```
  pos = pos + vel * 5;
  ```

  can be converted by the lexical analyzer (*scanner*) into:

  ```
  <id ,pos> <=> <id ,pos> <+> <id ,vel> <*> <int ,5> <;>
  ```

- In general whitespace, newlines and comments are not relevant in languages programming, so they can be eliminated by the lexical analyzer.

- Example 2: Geometric Processing Language Sketch:

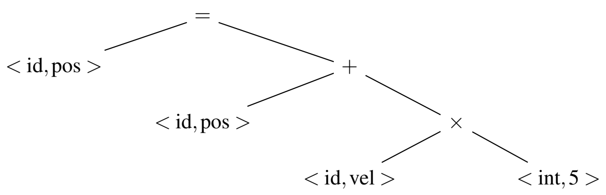
  ```
  distance ( 0 , 0 ) ( 4 , 3 )
  ```

  can be converted by the lexical analyzer (*scanner*) into:

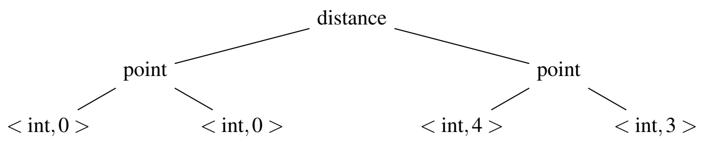
  ```
  <distance> <(> <num,0> <,> <num,0> <)>
  <(> <num,4> <,> <num,3> <)>
  ```

3.2 Syntax Analysis

- After the lexical analysis, the so-called syntactic analysis (*parsing*) is carried out, where conformity is verified of the sequence of lexical elements with the syntactic structure of the language.
- In languages that are intended to be syntactically processed, we can always make an approximation to the its formal structure through a representation like *tree*.
- For this purpose, a *grammar* is needed that specifies the desired structure (we will return to this problem later).
- In the example 1 (`pos = pos + vel * 5;`):



- In the example 2 (`distance ( 0 , 0 ) ( 4 , 3 )`):



- Attention is drawn to two characteristics of syntax trees:
  - does not include some lexical elements (which are only relevant to the formal structure);
  - unambiguously define the order of operations (we'll come back to this problem).

3.3 Semantic Analysis

- The final part of the compiler's *front end* is *semantic analysis*.
- In this phase, as much as possible, restrictions are checked that are not possible (or even desirable) to be made in the two previous phases.
- For example: check if an identifier was declared, check conformance in the type system of language, etc.
- Note that only constraints with static checking (i.e. at compile time), can be object of semantic analysis by the compiler.
- If in the example 2 there was an instruction for a circle of which the definition of its radius was part, it would not in general be possible, during semantic analysis, to guarantee a non-negative value for this radius (this semantics could only be verified dynamically, i.e., at runtime).

- Uses the parsing syntax tree as well as a data structure called symbol table (based on associative arrays).
- This last phase of analysis should guarantee the success of subsequent phases (generation and eventual code optimization, or interpretation).

3.4 Synthesis

- Having guaranteed that the source language code is valid, then we can move on to the intended effects with that code.
- Effects can be:
  1. simply the indication of source code validity;
  2. the translation of the source code into a target language;
  3. or interpretation and immediate execution.
- In all cases, it may be of interest to accurately identify and locate any errors.
- As most source code is text based, it is usual to indicate not only the instruction but also the line where each error occurs.

Code generation: example

- In the compilation process, it may be interesting to generate an intermediate representation of the code that facilitates the final generation of code.
- One possible form for this intermediate representation is the so-called *triple address code*.
- For example 1 (`pos = pos + vel * 5;`) we could have:

```
t1 = inttofloat(5)
t2 = id(vel) * t1
t3 = id(pos) + t2
id(pos) = t3
```

- This code could then be optimized in the next phase of compilation:

```
t1 = id(vel) * 5.0
id(pos) = id(pos) + t1
```

- And finally, one could generate *assembly* (pseudo-code):

```
LOAD   R2, id(vel)    // load value from memory to register R2
MULT   R2, R2, #5.0   // mult. 5 with R2 and store result in R2
LOAD   R1, id(pos)    // load value from memory to register R1
ADD    R1, R1, R2     // add R1 with R2 and store result in R1
STORE  id(pos), R1    // store value to memory from register R1
```

4 Implementation of a Compiler

To illustrate the work involved in language processors let's implement "by hand" an interpreter complete for the language suggested by the statement: `distance ( 0 , 0 ) ( 4 , 3 )`.

4.1 Lexical analysis

In order to "by hand" develop a lexical analyzer without excessive complications, we are going to oblige language *tokens* are separated by at least one white space and/or a newline. In this way, we can use the `Scanner` (methods `hasNext` and `next`) from the native library `Java`.

As a basic strategy to implement this parser, let's consider that it always has associated with it an actual *token*. For the start and end there will be two special *tokens*: `NONE` and `EOF`.

Each *token* has its type associated with it, and *tokens* of the same type share the same same lexical properties; and, when applicable, a (textual) attribute that completes its definition.

This parser will be usable by a method (`nextToken`) that will generate the next *token* consuming input characters.

The listing 1 shows a possible implementation of this program.

Compiling and running this program with input:

```
echo "distance ( 0 , 0 ) ( 1 , 4 )" | java -ea lexer/GeometryLanguageLexer
```

we get the following output:

```
<OP_DISTANCE> <OPEN_PARENTHESES> <NUMBER,0> <COMMA> <NUMBER,0> <CLOSE_PARENTHESES>
<OPEN_PARENTHESES> <NUMBER,1> <COMMA> <NUMBER,4> <CLOSE_PARENTHESES> <EOF>
```

Listing 1: Example lexical analyzer

```
package lexer;

import static java.lang.System.*;
import java.util.Scanner;

public class GeometryLanguageLexer {
   /**
    * token types
    */
   public enum tokenIds {
      NONE, NUMBER, COMMA, OPEN_PARENTHESES, CLOSE_PARENTHESES, DISTANCE, EOF
   }

   /**
    * Updates actual token to the next input token.
    */
   public static void nextToken() {
      assert token != tokenIds.EOF;

      token = tokenIds.EOF;
      attr = "";
      if (sc.hasNext()) {
         text = sc.next();
         switch(text) {
            case ",": token = tokenIds.COMMA; break;
            case "(": token = tokenIds.OPEN_PARENTHESES; break;
            case ")": token = tokenIds.CLOSE_PARENTHESES; break;
            case "distance": token = tokenIds.DISTANCE; break;
            default:
               attr = text;
               try {
                  value = Double.parseDouble(text);
                  token = tokenIds.NUMBER;
               }
               catch(NumberFormatException e) {
                  err.println("ERROR: unknown lexeme \""+text+"\"");
                  exit(1);
               }
               break;
         }
      }
   }

   /**
    * Actual token type
    */
   public static tokenIds token() { return token; }
   /**
    * Actual token attribute
    */
   public static String attr() { return attr; }
   /**
    * Actual token value
    */
   public static Double value() { return value; }

   public static void main(String[] args) {
      do {
         nextToken();
         out.print(" <"+token()+(attr().length() > 0 ? ","+attr() : "")+">");
      }
      while(token() != tokenIds.EOF);
      out.println();
   }

   protected static final Scanner sc = new Scanner(System.in);

   protected static tokenIds token = tokenIds.NONE;
   protected static String text = "";
   protected static String attr;
   protected static double value;

}
```

4.2 Syntax parsing

As a first approximation to building a parser let's make this just indicate whether the source code is a valid sequence of *tokens* (or not). To that end, we will follow the following strategy:

- Identify important language structures (rules);
- Associate boolean methods to the recognition of each rule;
- Ensure that, when invoking these methods, the *token* current is what you would expect at the beginning of these rules;
- The rule recognition process will have three possible behaviors:
  1. If successful, all tokens associated with the rule have been consumed (i.e. are part of of the lexical analyzer past);
  2. Failed for not recognizing the first *token* of the rule. In this case, there is no need to consume any token;
  3. Failed in the middle of rule recognition. In this situation, the parser just rejects the sequence of *tokens*.
- In this process, whenever a *token* is recognized, the parser will consume this *token* (i.e. advances to the next one).

The important structures (rules) in the presented sketch are the instruction *distance*. Since this instruction applies to colons, we also have the rule *period* (which in turn applies to a pair of numbers).

The listing 2 shows a possible implementation of this program.

4.3 Semantic analysis

The language defined so far does not allow for semantic errors that could serve as an example for this section. To solve this problem, let's add to the language the possibility to define and use *variables*. The existence of variables allows the existence of resulting semantic errors the use of undefined variables.

Variables are a programmatic feature that allows you to store values. using names (named *identifiers*). In this language, we will define an identifier as a non-empty sequence of lowercase letters (without accents).

The lexical analyzer has to be changed, adding the *tokens* `ID` and `EQUAL`:

```
...
   public enum tokenIds {
      NONE, NUMBER, COMMA, OPEN_PARENTHESES, CLOSE_PARENTHESES, DISTANCE, ID, EQUAL, EOF
   }
   public static void nextToken() {
...
            case "=": token = tokenIds.EQUAL; break;
            default:
               attr = text;
               if (attr.matches("[a-z]+"))
                  token = tokenIds.ID;
               else
               {
                  try {
                     value = Double.parseDouble(text);
                     token = tokenIds.NUMBER;
                  }
                  catch(NumberFormatException e) {
                     err.println("ERROR: unknown lexeme \""+text+"\"");
                     exit(1);
                  }
               }
               break;
...
   }
...
```

The proper data structure for dealing with variables is the associative *array*. With this data structure, we can associate the values we want to the variable name. For now we just want to know if the variable is or is not defined. Therefore, the existence of the identifier in the associative *array* is sufficient.

Listing 2: Example parser

```
package parser;

import static java.lang.System.*;
import static lexer.GeometryLanguageLexer.*;

public class GeometryLanguageParser {
   /**
    * Start rule: attempts to parse the whole input.
    */
   public static boolean parse() {
      assert token() == tokenIds.NONE;

      nextToken();
      return parseDistance();
   }

   /**
    * Distance rule parsing.
    */
   public static boolean parseDistance() {
      assert token() != tokenIds.NONE;

      boolean result = token() == tokenIds.DISTANCE;
      if (result) {
         nextToken();
         result = parsePoint();
         if (result) {
            result = parsePoint();
         }
      }
      return result;
   }

   /**
    * Point rule parsing.
    */
   public static boolean parsePoint() {
      assert token() != tokenIds.NONE;

      boolean result = token() == tokenIds.OPEN_PARENTHESES;
      if (result) {
         nextToken();
         result = token() == tokenIds.NUMBER;
         if (result) {
            nextToken();
            result = token() == tokenIds.COMMA;
            if (result) {
               nextToken();
               result = token() == tokenIds.NUMBER;
               if (result) {
                  nextToken();
                  result = token() == tokenIds.CLOSE_PARENTHESES;
                  if (result) {
                     nextToken();
                  }
               }
            }
         }
      }
      return result;
   }

   public static void main(String[] args) {
      if(parse())
         out.println("Ok");
      else
         out.println("ERROR");
   }

}
```

Listing 3: Expressions.

```
public static boolean parseExpression() {
   assert token() != tokenIds.NONE;

   boolean result = true;
   switch(token()) {
      case NUMBER:
         nextToken();
         break;
      case ID:
         if (!symbolTable.containsKey(attr()))
         {
            err.println("ERROR: undefined variable \""+attr()+"\"");
            exit(1);
         }
         nextToken();
         break;
      default:
         result = parseDistance();
         break;
   }
   return result;
}
```

Listing 4: Value assignment.

```
public static boolean parseAssignment() {
   assert token() != tokenIds.NONE;

   boolean result = token() == tokenIds.ID;
   if (result) {
      String var = attr();
      nextToken();
      result = token() == tokenIds.EQUAL;
      if (result) {
         nextToken();
         result = parseExpression();
         if (result) {
            symbolTable.put(var, null);
         }
      }
   }
   return result;
}
```

```
protected static Map<String,Object> symbolTable = new HashMap<String,Object>();
```

Let's also generalize the language a bit by defining the concept of *expression*. An expression will be a program entity that has a numerical value associated with it. In this case, it could be a literal number, a variable or the distance instruction. The code 3 presents a method that does this parsing.

To enable this generalization, simply replace the literal number parsing (`NUMBER`) by an invocation of this new method. Note that this new syntactic structure greatly increases the flexibility of the language, since now where you expect a numeric value (point coordinate, value assignment) any expression can appear (instead of just a literal number).

We now need to add a value assignment statement (which sets, or resets, the value of a variable). The code 4 shows this method.

To make the new statement active, let's modify the initial parsing method (code 5).

4.4 Synthesis: code interpretation

To complete this example, it only remains to implement actions linked to the defined language. In order not to complicate the problem, let's assume that all instructions have a numerical value, and that the effect

Listing 5: New method *parse*.

```
   public static boolean parse() {
      assert token() == tokenIds.NONE;

      nextToken();
      boolean result = true;
      while(result && token() != tokenIds.EOF) {
         result = parseDistance();
         if (!result)
            result = parseAssignment();
      }
      return result;
   }
```

of an instruction is the writing of that value. So, for example, the application of the distance instruction must calculate and write this value.

To implement this behavior we will insert the necessary code directly into the parser to carry out these actions. As instructions become associated with numerical values, we need to find a way to associate these values. In that sense, let's replace the boolean results with the non-primitive data type `Double`. A result equal to **`null`** indicates syntactic error, and a non-null value, expresses the value associated with the instruction. The only exception to this procedure will be the parsing of points as they have to be associated to a pair of values (not just one).

The code 6 exemplifies the code of the interpreter (the complete program is provided in annex).

5 Languages: Definition as a Set

- Languages are for *communicating*.
- A message can be seen as a sequence of *symbols*.
- However, a language does not accept all types of symbols and sequences.
- A language is characterized by a set of symbols and a way of describe valid sequences of these symbols (i.e. the set of valid sequences).
- If natural languages allow for some subjectivity and ambiguity, programming languages require complete objectivity.

- How to define languages in a synthetic and objective way?
- Defining by *extension* – that is, enumerating all possible occurrences – is one possibility.
- However, for minimally interesting not only would we have a gigantic description, but also probably an incomplete one.
- Programming languages tend to accept infinite variants of input.
- Alternatively we can describe it as *understanding*.
- One possibility is to use the formalisms linked to the definition of *sets*.

5.1 Basic concepts and terminology

- A set can be defined by *extension* (or enumeration) or by *comprehension*.
- An example of a set defined by extension is the set of binary digits $\{0,1\}$.
- In the definition by comprehension, the following notation is used:

$$\{x \mid p(x)\}$$

  or

$$\{x : p(x)\}$$

- x is the variable that represents any element of the set, and $p(x)$ is a predicate on that variable.
- Thus, this set is defined as containing all values of x where the predicate $p(x)$ is true.
- For example: $\{n \mid n \in \mathbb{N} \wedge n \leq 9\} \quad = \quad \{1,2,3,4,5,6,7,8,9\}$

Listing 6: Interpretador.

```
...
   public static Double parseAssignment() {
      assert token() != tokenIds.NONE;

      Double result = null;
      if (token() == tokenIds.ID) {
         String var = attr();
         nextToken();
         if (token() == tokenIds.EQUAL) {
            nextToken();
            result = parseExpression();
            if (result != null) {
               symbolTable.put(var, result);
            }
         }
      }
      return result;
   }
...
   public static Double parseDistance() {
      assert token() != tokenIds.NONE;

      Double result = null;
      if (token() == tokenIds.DISTANCE) {
         nextToken();
         Double[] p1 = parsePoint();
         if (p1 != null) {
            Double[] p2 = parsePoint();
            if (p2 != null) {
               result = Math.sqrt(Math.pow(p1[0]-p2[0],2)+Math.pow(p1[1]-p2[1],2));
            }
         }
      }
      return result;
   }
...
   public static Double[] parsePoint() {
      assert token() != tokenIds.NONE;

      Double[] result = null;
      if (token() == tokenIds.OPEN_PARENTHESES) {
         nextToken();
         Double x = parseExpression();
         if (x != null) {
            if (token() == tokenIds.COMMA) {
               nextToken();
               Double y = parseExpression();
               if (y != null) {
                  if (token() == tokenIds.CLOSE_PARENTHESES) {
                     nextToken();
                     result = new Double[2];
                     result[0] = x;
                     result[1] = y;
                  }
               }
            }
         }
      }
      return result;
   }
...
```

- A *symbol* (or *letter*) is the atomic (indivisible) unit of languages.
- In text-based languages, a symbol will be a character.
- An *alphabet* is a non-empty finite set of symbols.
- For example:
  - $A = \{0,1\}$ is the alphabet of binary digits.
  - $A = \{0,1,\cdots,9\}$ is the alphabet of decimal digits.
- A *word* (*string* or string) is a sequence of symbols about a given alphabet A.

$$U = a_1 a_2 \cdots a_n, \quad \text{com} \quad a_i \in A \wedge n \geq 0$$

- For example:
  - $A = \{0,1\}$ is the alphabet of binary digits.
    $01101, 11, 0$
  - $A = \{0,1,\cdots,9\}$ is the alphabet of decimal digits.
    $2016, 234523, 999999999999999, 0$
  - $A = \{0,1,\cdots,0,a,b,\cdots,z,@,\cdots\}$
    $\text{mos@ua.pt}, \text{Good morning!}$
- The *empty word* is a sequence of zero symbols and is denoted by ε (epsilon).
- Note that ε does not belong to the alphabet.
- An *subword* of a word u is a contiguous string of 0 or more u symbols.
- A *prefix* of a word u is a contiguous string of 0 or more leading symbols of u.
- An *suffix* of a word u is a contiguous sequence of 0 or more terminal symbols of u.
- For example:
  - as is a subword of home, but not a prefix or suffix
  - 001 is prefix and subword of 00100111 but not suffix
  - ε is prefix, suffix and subword of any word u
  - any word u is prefix, suffix and subword of itself
- The *closing* (or set of strings) of the alphabet A named by A^*, represents the set of all the definable words over the A alphabet, including the empty word.
- For example:
  - $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \cdots\}$
  - $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}^* = \{\varepsilon, \clubsuit, \diamondsuit, \heartsuit, \spadesuit, \clubsuit\diamondsuit, \cdots\}$
- Given an alphabet A, a *language* L over A is a finite or infinite set of valid words defined with A symbols.
  That is: $L \subseteq A^*$
- Example languages about the alphabet $A = \{0,1\}$
  - $L_1 = \{u \mid u \in A^* \wedge |u| \leq 2\} = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$
  - $L_2 = \{u \mid u \in A^* \wedge \forall_i u_i = 0\} = \{\varepsilon, 0, 00, 000, 0000, \cdots\}$
  - $L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1) \mathit{mod}\, 2 = 0\} = \{000, 11, 000110101, \cdots\}$
  - $L_4 = \{\} = \emptyset$ (emptyset)
  - $L_5 = \{\varepsilon\}$
  - $L_6 = A$
  - $L_7 = A^*$
- Note that $\{\}$, $\{\varepsilon\}$, A and A^* are languages over the alphabet A whatever A is
- Since languages are sets, all mathematical operations on sets apply: meeting, interception, complement, difference, etc.

5.2 Operations on words

- The *length* of a word u is denoted by $|u|$ and represents its number of symbols.
- The length of the empty word is zero

$$|\varepsilon| = 0$$

- It is customary to interpret the word u as a function to access its symbols (like *array*):

$$u : \{1,2,\cdots,n\} \rightarrow A, \quad \text{com} \quad n = |u|$$

  where u_i represents the ith symbol of u
- The *reverse* of a word u is the word, denoted by u^R, and is obtained reversing the order of u symbols

$$u = \{u_1, u_2, \cdots, u_n\} \implies u^R = \{u_n, \cdots, u_2, u_1\}$$

- The *concatenation* (or *product*) of the words u and v is denoted by $u.v$, or simply uv, and represents the juxtaposition of u and v, i.e., the word consisting of the u symbols followed by the v symbols.
- Concatenation properties:
  - $|u.v| = |u| + |v|$
  - $u.(v.w) = (u.v).w = u.v.w$ (associative)
  - $u.\varepsilon = \varepsilon.u = u$ (neutral element)
  - $u \neq \varepsilon \wedge v \neq \varepsilon \wedge u \neq v \implies u.v \neq v.u$ (non-commutative)
- The *power* of order n, with $n \geq 0$, of a word u is denoted by u^n and represents the concatenation of n replicas of u, that is, $\underbrace{uu\cdots u}_{n\times}$.
- $u^0 = \varepsilon$

5.3 Operations on languages

Language operations: meeting

- The *meeting* of two languages L_1 and L_2 is denoted by $L_1 \cup L_2$ and is given by:

$$L_1 \cup L_2 = \{u \mid u \in L_1 \vee u \in L_2\}$$

- For example, if we define the languages L_1 and L_2 over the alphabet $A = \{a,b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ starts with } The\} = \{aW \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ ends with } The\} = \{wa \mid w \in A^*\} \end{aligned}$$

- what will be the result of merging these languages?

$$L = L_1 \cup L_2 = ?$$

- Answer:

$$L = \{w_1 a w_2 \mid w_1, w_2 \in A^* \wedge (w_1 = \varepsilon \vee w_2 = \varepsilon)\}$$

Language operations: interception

- The *interception* of two languages L_1 and L_2 is denoted by $L_1 \cap L_2$ and is given by:

$$L_1 \cap L_2 = \{u \mid u \in L_1 \wedge u \in L_2\}$$

- For example, if we define the languages L_1 and L_2 over the alphabet $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ starts with } The\} = \{aW \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ ends with } The\} = \{wa \mid w \in A^*\} \end{aligned}$$

- what will be the result of the interception of these languages?

$$L = L_1 \cap L_2 = ?$$

- Answer:

$$L = \{awa \mid w \in A^*\} \cup \{The\}$$

Operations on languages: difference

- The *difference* of two languages L_1 and L_2 is denoted by $L_1 - L_2$ and is given by:

$$L_1 - L_2 = \{u \mid u \in L_1 \wedge u \notin L_2\}$$

- For example, if we define the languages L_1 and L_2 over the alphabet $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ starts with } The\} = \{aW \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ ends with } The\} = \{wa \mid w \in A^*\} \end{aligned}$$

- what will be the result of the difference of these languages?

$$L = L_1 - L_2 = ?$$

- Answer:

$$L = \{awx \mid w \in A^* \wedge x \in A \wedge x \neq a\}$$

- or:

$$L = \{awb \mid w \in A^*\}$$

Operations on languages: completion

- The *complement* of the L language is denoted by $\overline{L}$ and is given by:

$$\overline{L} = A^* - L = \{u \mid u \notin L\}$$

- For example, if we define the language L_1 over the alphabet $A = \{a, b\}$:

$$L_1 \;=\; \{u \mid u \text{ starts with } The\} \;=\; \{aW \mid w \in A^*\}$$

- what will be the result of complementing this language?

$$L \;=\; \overline{L_1} \;=?$$

- Answer:

$$L \;=\; \{xw \mid w \in A^* \;\wedge\; x \in A \;\wedge\; x \neq a\} \;\cup\; \{\varepsilon\}$$

- or:

$$L \;=\; \{bw \mid w \in A^*\} \;\cup\; \{\varepsilon\}$$

Language operations: concatenation

- The *concatenation* of two languages L_1 and L_2 is denoted by $L_1.L_2$ and is given by:

$$L_1.L_2 = \{uv \mid u \in L_1 \;\wedge\; v \in L_2\}$$

- For example, if we define the languages L_1 and L_2 over the alphabet $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ starts with } The\} \;=\; \{aW \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ ends with } The\} \;=\; \{wa \mid w \in A^*\} \end{aligned}$$

- what will be the result of the concatenation of these languages?

$$L \;=\; L_1.L_2 \;=?$$

- Answer:

$$L \;=\; \{awa \mid w \in A^*\}$$

Operations on languages: potentiation

- The *power* of order n of the language L is denoted by L^n and is defined inductively by:

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^{n+1} &= L^n.L \end{aligned}$$

- For example, if we define the language L_1 over the alphabet $A = \{a, b\}$:

$$L_1 \;=\; \{u \mid u \text{ starts with } The\} \;=\; \{aW \mid w \in A^*\}$$

- what will be the result of the power of order 2 of this language?

$$L \;=\; L_1^2 \;=?$$

- Answer:

$$L \;=\; \{aw_1aw_2 \mid w_1, w_2 \in A^*\}$$

Operations on languages: Kleene closure

- The *Kleene closure* of the language L is denoted by L^* and is given by:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \bigcup_{i=0}^{\infty} L^i$$

- For example, if we define the language L_1 over the alphabet $A = \{a, b\}$:

$$L_1 = \{u \mid u \text{ starts with } The\} = \{aW \mid w \in A^*\}$$

- what will be the Kleene closure of this language?

$$L = L_1^* = ?$$

- Answer:

$$L = L_1 \cup \{\varepsilon\}$$

- Note that for $n > 1 \quad L_1^n \subset L_1$

Language operations: additional notes

- Note that in binary operations on sets it is not required that the two languages are defined on the same alphabet.
- So if we have two languages L_1 and L_2 respectively defined on the alphabets A_1 and A_2, then the alphabet resulting from the application of any binary operation on the languages is: $A_1 \cup A_2$

6 Introduction to grammars

- The use of sets to define languages is often not the most appropriate way and versatile to describe them.
- It is often preferable to identify intermediate structures, which abstract parts or important subsets of the language.
- As in programming, many times recursive descriptions are much simpler, without losing the necessary objectivity and rigor.
- This is where we find the *grammars*.
- *grammars* describe languages by comprehension using *formal* and (often) *recursive* representations.
- Seeing languages as sequences of symbols (or words), grammars formally define the *valid* sequences.

- For example, in Portuguese the sentence “The dog barks” can be grammatically described by:

$$\begin{aligned}
\text{phrase} &\to \text{subject predicate}\\
\text{subject} &\to \text{article noun}\\
\text{predicate} &\to \text{verb}\\
\text{article} &\to \textbf{O} \mid \textbf{Um}\\
\text{noun} &\to \textbf{dog} \mid \textbf{wolf}\\
\text{verb} &\to \textbf{thief} \mid \textbf{howl}
\end{aligned}$$

- This (non-recursive) grammar formally describes 8 possible sentences, which is still uninteresting.
- However, it contains more information than the original sentence, as it sorts the various elements of the sentence (subject, predicate, etc.).
- Contains 6 *terminal symbols* and 6 *non-terminal symbols*.
- A non-terminal symbol is defined by an *production* describing possible representations of that symbol, depending on terminal and/or non-terminal symbols.

- Formally, a grammar is a quadruple $G = (T, N, S, P)$, where:
  1. T is a non-empty finite set called the terminal alphabet, where each element is called the *terminal* symbol;
  2. N is a non-empty finite set, disjoint from T ($N \cap T = \emptyset$), whose elements are designated by *non-terminal* symbols;
  3. $S \in N$ is a specific non-terminal symbol called *start symbol*;
  4. P is a finite set of *rules* (or productions) of the form $\alpha \to \beta$ where $\alpha \in (T \cup N)^* N (T \cup N)^*$ and $\beta \in (T \cup N)^*$, that is, α is a string of terminal and non-terminal symbols containing at least one non-terminal symbol; and β is a string of symbols, eventually empty, terminal and non-terminal.

Grammars: examples

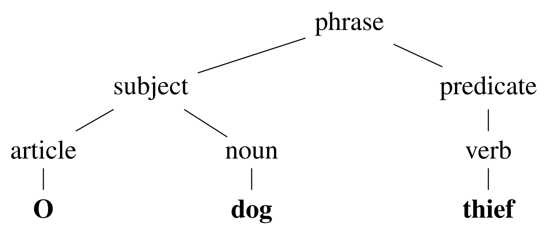
- Formally, the previous grammar will be:

$$\begin{aligned} G \;=\; & (\{\mathbf{O}, \mathbf{Um}, \mathbf{dog}, \mathbf{wolf}, \mathbf{thief}, \mathbf{howl}\}, \\ & \{\text{sentence}, \text{subject}, \text{predicate}, \text{article}, \text{noun}, \text{verb}\}, \\ & \text{phrase}, P) \end{aligned}$$

- P is made up of the rules already presented:

$$\begin{aligned} \text{phrase} &\to \text{subject predicate} \\ \text{subject} &\to \text{article noun} \\ \text{predicate} &\to \text{verb} \\ \text{article} &\to \mathbf{O} \mid \mathbf{A} \\ \text{noun} &\to \mathbf{dog} \mid \mathbf{wolf} \\ \text{verb} &\to \mathbf{thief} \mid \mathbf{howl} \end{aligned}$$

- We can describe the sentence "The dog barks" with the following tree (called syntactic).



- Consider the following grammar $G = (\{0, 1\}, \{S, A\}, S, P)$, where P consists of the rules:

$$\begin{aligned} S &\to 0S \\ S &\to 0A \\ A &\to 0A1 \\ A &\to \varepsilon \end{aligned}$$

- What will be the language defined by this grammar?

$$L = \{0^n1^m : n \in \mathbb{N} \wedge m \in \mathbb{N}_0 \wedge n > m\}$$

- Being $A = \{a, b\}$, define a grammar for the following language:

$$L_1 = \{aW \mid w \in A^*\}$$

- The grammar $G = (\{a,b\}, \{S,X\}, S, P)$, where P is made up of the rules:

$$\begin{aligned} S &\to aX \\ X &\to aX \\ X &\to bX \\ X &\to \varepsilon \end{aligned}$$

or:

$$\begin{aligned} S &\to aX \\ X &\to aX \mid bX \mid \varepsilon \end{aligned}$$

- Being $A = \{0, 1\}$, define a grammar for the following language:

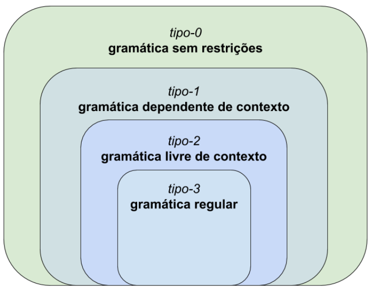
$$L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1)\ mod\ 2 = 0\}$$

- The grammar $G = (\{0,1\}, \{S,A\}, S, P)$, where P is made up of the rules:

$$\begin{aligned} S &\to S\,1\,S\,1\,S \mid A \\ A &\to 0\,A \mid \varepsilon \end{aligned}$$

6.1 Chomsky hierarchy

- Constraints on α and β allow to define a taxonomy of languages – Chomsky hierarchy:
  1. If there is no restriction, G is designated by *type*-0 grammar.
  2. G will be of 1-*type*, or *context-dependent* grammar, if each rule $\alpha \to \beta$ of P obeys $|\alpha| \leq |\beta|$ (with the exception that there may also be the empty output: $S \to \varepsilon$).
  3. G will be of 2-*type*, or grammar *context-independent, or free*, if each rule $\alpha \to \beta$ of P obeys $|\alpha| = 1$, that is: α consists of a single non-terminal.
  4. G will be of 3-*type*, or *regular* grammar, if each rule has one of the forms: $A \to cB$, $A \to c$ or $A \to \varepsilon$, where A and B are non-terminal symbols (A can be equal to B) and c a terminal symbol. That is, in all productions, β can only have at most one non-terminal symbol always on the right (or, alternatively, always on the left).



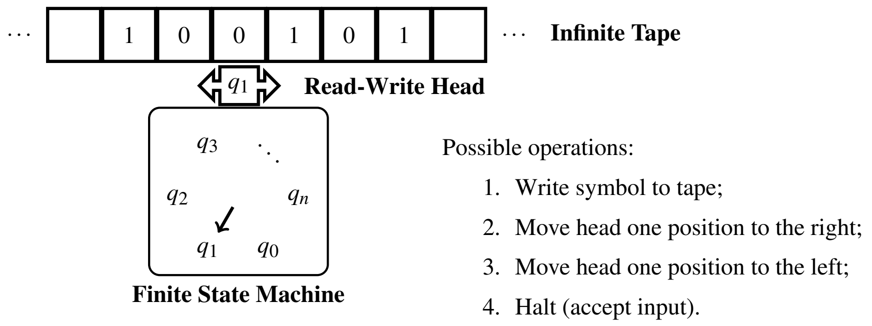
- For each of these types different types of machines can be defined (algorithms, automata) that can recognize them.
- The simpler the grammar, the simpler and more efficient the machine that recognizes these languages.

- Each language class of *type-i* contains the language class *type*-$(i+1)$ $(i = 0,1,2)$
- This hierarchy not only translates the formal characteristics of languages, but also expresses The required computing requirements:
  1. *Turing machines* process grammars without restrictions (type-0);
  2. *linearly bounded automata* process context-dependent (1-type) grammars;
  3. *Pushdown automata* process context-independent grammars (2-type);
  4. *finite automata* process regular grammars (3-type).

6.2 Automatons

6.2.1 Turing Machine

- (Alan Turing, 1936)
- Abstract computing model.
- Allows (in theory) to implement any computable program.
- Relies on a finite state machine, symbol read/write head and on an infinite tape (where these symbols are written or read).
- The read/write "head" can move one position left or right.
- Very important model in the theory of computation.
- Little relevant in the practical implementation of language processors.



Possible operations:

1. Write symbol to tape;
2. Move head one position to the right;
3. Move head one position to the left;
4. Halt (accept input).

- The finite state machine (FSM) has access to the current symbol and decides the next action to be performed.
- The action consists of the state transition and what is the operation on the tape.
- If no action is possible, the input is rejected.

Turing Machine: example

- Given the alphabet $A = \{0,1\}$, and considering that a non-negative integer n is represented by sequence of $n+1$ 1 symbols, let's implement a TM that adds the next ones (i.e. to the right of the current position) two existing integers on the tape (separated only by a 0).
- The algorithm could be simply swapping the 0 symbol between the two numbers for 1, and swapping the last two symbols 1 for 0.
- For example: $3+2$ which corresponds to the following state on the tape (bold symbol is the position of the "head"): $\cdots\mathbf{0}111101110\cdots$ (the intended result will be: $\cdots\mathbf{0}111111000\cdots$).
- Whereas the states are designated by $E_i, i \geq 1$ (E_1 being the initial state); and the operations:

  d move one position to the right;

  e move one position to the left;

0 write the 0 symbol on the tape;

1 write the 1 symbol on the tape;

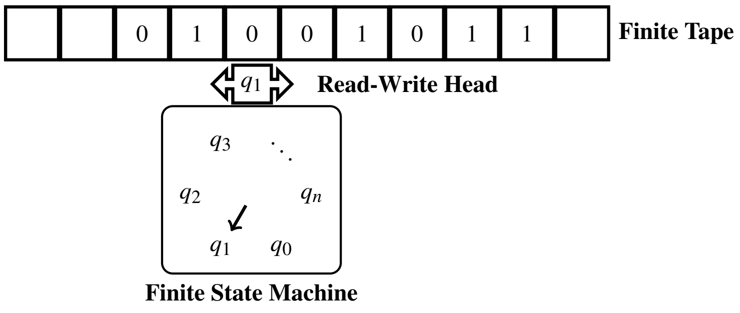
h accept and terminate automaton.

- A possible solution is given by the following state transition diagram:

	Input	
State	**0**	**1**
E_1	E_1/d	E_2/d
E_2	$E_3/1$	E_2/d
E_3	E_4/e	E_3/d
E_4	$--$	$E_5/0$
E_5	E_5/e	$E_6/0$
E_6	E_7/e	$--$
E_7	E_1/h	E_7/e

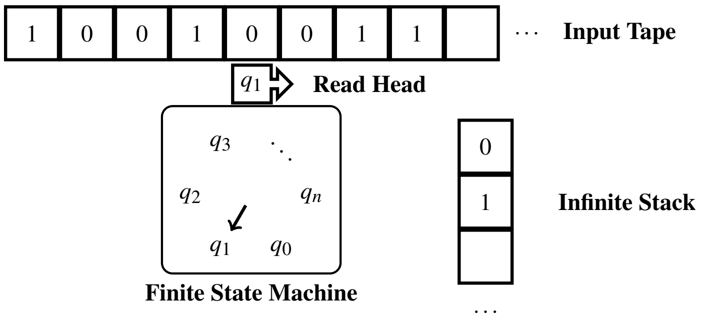
- $E_1 \cdots \mathbf{0}111101110 \cdots \rightarrow E_1 \cdots 0\mathbf{1}11101110 \cdots \xrightarrow{*} E_2 \cdots 01111\mathbf{0}1110 \cdots \rightarrow E_3 \cdots 01111\mathbf{1}1110 \cdots \rightarrow E_3 \cdots 011111\mathbf{1}110 \cdots \xrightarrow{*} E_3 \cdots 011111111\mathbf{0} \cdots$ $\rightarrow E_4 \cdots 01111111\mathbf{1}0 \cdots \rightarrow E_5 \cdots 01111111\mathbf{0}0 \cdots \rightarrow E_5 \cdots 0111111\mathbf{1}00 \cdots \rightarrow E_6 \cdots 0111111\mathbf{0}00 \cdots \rightarrow E_7 \cdots 011111\mathbf{1}000 \cdots \xrightarrow{*}$ $E_7 \cdots \mathbf{0}111111000 \cdots$

6.2.2 Linearly bounded automata



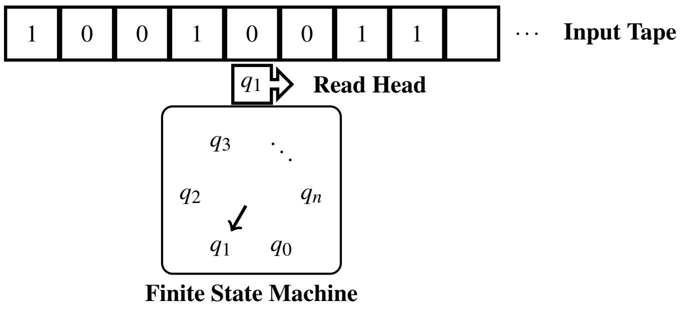
- They differ from MT by the finitude of the tape.

6.2.3 Pushdown automata



- "Head" is read-only and supports an unbounded stack.
- Movement of the "head" in one direction only.
- Automata suitable for parsing.

6.2.4 Finite automata



- No state machine write support.
- Automata suitable for lexical analysis.