Theme 1

Compilers, Languages and Grammars

Introduction

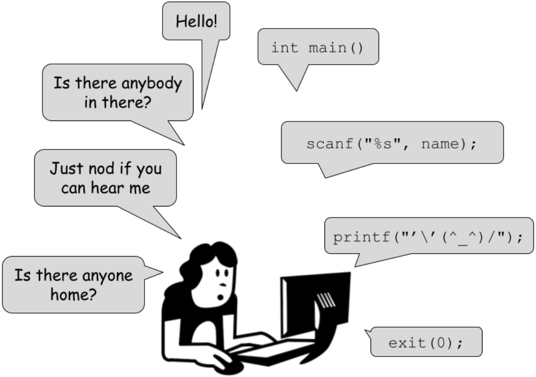
Compilers, 2nd semester 2024-2025

Miguel Oliveira e Silva
DETI, University of Aveiro, Portugal

Framework

Frame

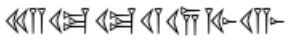
- In this course we will talk about languages – what they are and how we can define them – and about compilers – tools that recognize them and allow you to perform actions as a result of that process.



A comfortably numb … programmer!

Frame (2)

- If you had to define language how would you do it?
  - Allows express, transmit and receive ideas.
  - Communication between people or living beings in general.
  - Includes communication with and between machines.
  - Requires several communicating entities, a code and rules for the communication to be intelligible.
  - Requires the use of (at least) one medium or communication channel (sound, text, . . .).
- Required: coding and a set of common rules, and interlocutors who know them.
- Let's look at some natural languages as an example.
- Different words, in different languages, can have the same meaning:
  - "*adeus*", "*goodbye*", "*au revoir*",



- On the other hand, there are also words that are the same with different meanings (depending on the context):
  - *hill*, *river*, *path*, ...

Frame (3)

- 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.
- 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:

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

    - 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.

Frame (4)

- Inherent in languages is the need to decide whether a sequence of alphabet symbols is valid.
  - **correct**:
    a + d + e + u + s $\rightarrow$ goodbye
    goodbye + and + until + tomorrow $\rightarrow$
    goodbye and see you tomorrow
  - **incorrect**:
    e + d + u + a + s $\rightarrow$ edes
    until+goodbye+tomorrow+and $\rightarrow$ 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.

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!

Compilers: Introduction

Compilers

Comprehension, interpretation and/or automatic translation of languages.

Compilers (Language Processors)

- 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.

source language ⟶ `Compiler` ⟶ target language

- 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:

input ⟶ `Program` ⟶ output

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

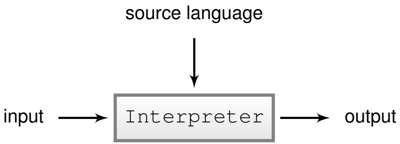
- Source code:
- ```
  1+2*3:4
  ```

- A possible compilation for Java:

```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);
   }
}
```
```

Compilers: Language Processors (2)

- 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

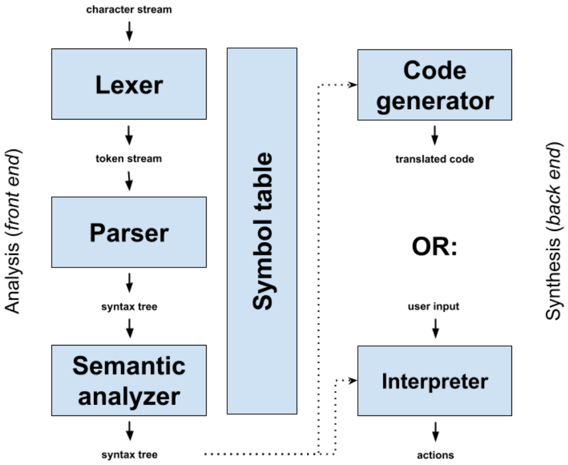
- Source code:
-
```
1+2*3:4
```

- One possible interpretation:

```
2.5
```

Structure of a Compiler

Structure of a Compiler



Structure of a Compiler (2)

- 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.

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> <;>
```

Lexical Analysis (2)

- 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:

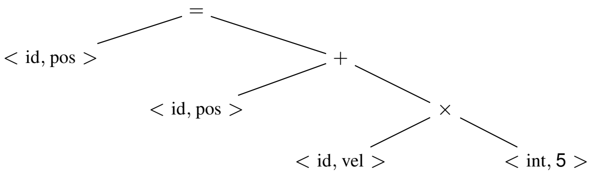
```
distance ( 0 , 0 ) ( 4 , 3 )
```

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

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

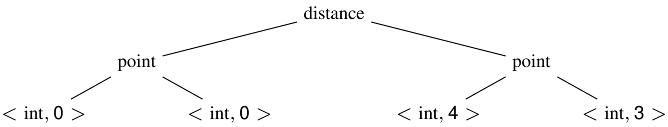
Syntax Analysis

- 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;`):



Syntax Analysis (2)

- 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).

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).

Semantic Analysis (2)

- 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).

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
```

Languages: Definition as Set

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.

Languages (2)

- How to define languages in a synthetic and objective way?
- Setting by extension is a 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.

Basic Concepts

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:
$\{n \mid n \in \mathbb{N} \land n \leq 9\} \quad = \quad \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Basic concepts and terminology (2)

- 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.

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

Basic concepts and terminology (3)

- 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.
    $2016, 234523, 999999999999999, 0$
  - $A = \{0, 1, \cdots, 0, a, b, \cdots, z, @, \cdots\}$
    mos@ua.pt, Good morning!

Basic concepts and terminology (4)

- 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

Basic concepts and terminology (5)

- 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: $L \subseteq A^*$

Basic concepts and terminology (6)

- Example languages about the alphabet $A = \{0, 1\}$
  - $L_1 = \{u \mid u \in A^* \wedge |u| \leq 2\} = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$
  - $L_2 = \{u \mid u \in A^* \wedge \forall_i\, u_i = 0\} = \{\varepsilon, 0, 00, 000, 0000, \cdots\}$
  - $L_3 = \{u \mid u \in A^* \wedge u.\mathrm{count}(1)\ mod\ 2 = 0\} = \{000, 11, 000110101, \cdots\}$
  - $L_4 = \{\} = \emptyset$ (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.

Operations on words

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

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

Operations on words (2)

- 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, $\underbrace{uu \cdots u}_{n\times}$.
- $u^0 = \varepsilon$

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:

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

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

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

- what will be the result of merging these languages?

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

- Answer:

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

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 $A = \{a, b\}$:

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

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

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

- Answer:

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

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 $A = \{a, b\}$:

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

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

$$L = L_1 - L_2 = ?$$

- Answer:

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

- or:

$$L = \{a\,w\,b \mid w \in A^*\}$$

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:

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

- or:

$$L = \{b\, w \mid w \in A^*\} \cup \{\varepsilon\}$$

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 $A = \{a, b\}$:

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

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

$$L = L_1.L_2 = ?$$

- Answer:

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

Operations on languages: potentiation

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

$$L^0 = \{\varepsilon\}$$
$$L^{n+1} = L^n.L$$

- 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 the power of order 2 of this language?

$$L = L_1^2 = ?$$

- Answer:

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

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: $A_1 \cup A_2$

Introduction to grammars

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.

Grammars (2)

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

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

- This grammar describes 8 possible sentences and contains more information than the original sentence.
- 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.

Introduction to grammars (2)

- 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:

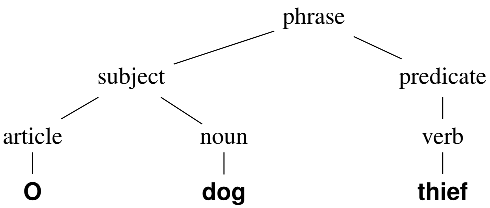
$$G = (\{\textbf{O}, \textbf{Um}, \textbf{dog}, \textbf{wolf}, \textbf{thief}, \textbf{howl}\},$$
$$\{\text{sentence}, \text{subject}, \text{predicate}, \text{article}, \text{noun}, \text{verb}\},$$
$$\text{phrase}, P)$$

- P is made up of the rules already presented:

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

Grammars: examples (2)

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



Grammars: examples (3)

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

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

- What will be the language defined by this grammar?

$$L = \{0^n 1^m : n \in \mathbb{N} \wedge m \in \mathbb{N}_0 \wedge n > m\}$$

Grammars: examples (4)

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

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

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

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

or:

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

Grammars: examples (5)

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

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

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

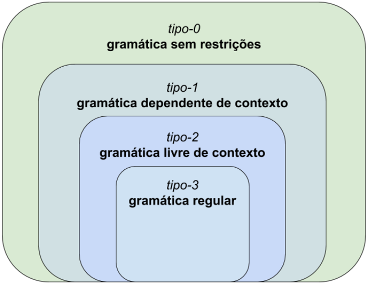
$$\begin{aligned} S &\rightarrow\; S\,1\,S\,1\,S \;|\; A \\ A &\rightarrow\; 0\,A \;|\; \varepsilon \end{aligned}$$

Chomsky Hierarchy

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 $\alpha \rightarrow \beta$ of P obeys $|\alpha| \leq |\beta|$ (with the exception that there may also be the empty output: $S \rightarrow \varepsilon$).
    3. G will be of 2-type, or grammar context-independent, or free, if each rule $\alpha \rightarrow \beta$ 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: $A \rightarrow cB$, $A \rightarrow c$ or $A \rightarrow \varepsilon$, 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).

Chomsky hierarchy (2)



- 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.

Chomsky hierarchy (3)

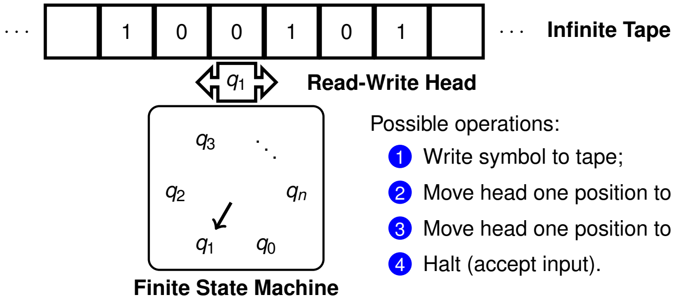
- Each language class of type-i contains the language class type-$(i+1)$ ($i = 0, 1, 2$)
- This hierarchy not only translates the formal characteristics of languages, but also expresses The required computing requirements:
  1. Turing machines process grammars without restrictions (type-0);
  2. linearly 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).

Automata

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.

Turing Machine (2)



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.

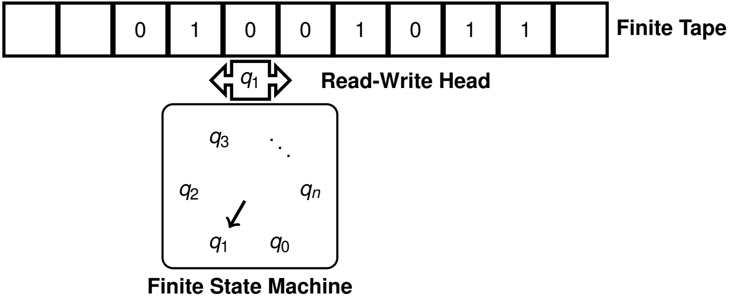
Turing machine: example (2)

- 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

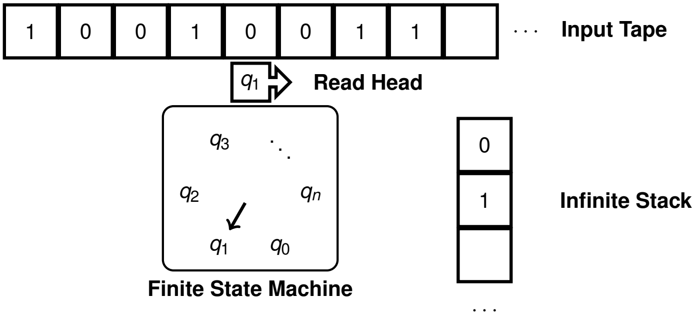
- $$E_1 \cdots \mathbf{0}111101110 \cdots \rightarrow E_1 \cdots 0\mathbf{1}11101110 \cdots \xrightarrow{*} E_2 \cdots 01111\mathbf{0}1110 \cdots \rightarrow$$
  $$E_3 \cdots 01111\mathbf{1}1110 \cdots \rightarrow E_3 \cdots 011111\mathbf{1}110 \cdots \xrightarrow{*} E_3 \cdots 011111111\mathbf{0} \cdots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

Linearly bounded automata



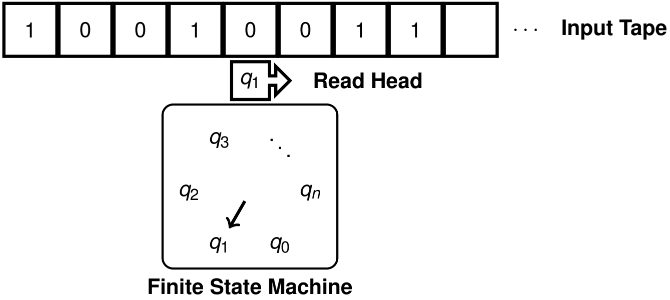
- They differ from MT by the finitude of the tape.

Pushdown automata



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

Finite automata



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