Houssam Saber

Language & Complete

Notes & Additional Materials

DATE OF SUBMISSION

Lectures given by Dr. Violetta Cavalli-Sforza

Contents

List of Figures

Chapter 1

Introduction to Programming Languages

1.1 Intro to PLs: Outline & Objectives

1.1.1 Why study PLs ?

We can see CS issues as design issues with PL design issues, an example is an interactive application where the user has to input something, and you expect them t use certain symbols, and that's a problem of PL design itself. Learning PLs would help in improving your ability to:

- develop effective algorithms.
- Use an existing PL.
- Understand useful programming constructs in PLs.
- Learn new PLs.
- Evaluate suitability of a PL for different tasks according to different criteria.

1.1.2 Languages: Natural & Formal

Natural languages are like arabic and french and english and spanish, formal languages are like mathematics and logic.

1.1.3 Synatax & Semantics

Syntax is the structure of a language, and semantics is the meaning of a language. A sentence may be gramatically correct but semantically incorrect, example: "The house reads a book." The meaning of the whole is the meaning of the parts. This is true in this example above and always in programming. This excludes figurative language.

1.1.4 What is a PL?

- A formal language whose purpose is special and language limited.
- Set of rules and symbols used to construct a computer program.
- A language used to interact with the computer
- Capable of expressing any computer programming.
- Equivalent to a universal turing machine. All PLs are equivalent in this sense.

A PL has:

- Like a natural language (NL) :
  - Words (lexemes) (Reserved, operators, +, -, *) and punctuation(() , ; ,{ } , []).
  - Synatax
  - Semantics
- Unlike a NL, a PL has NO ambiguity. (Everything means just one thing.)

1.2 Factors affecting PL evolution

1.2.1 Applications

- 1960s:
  - Business Processes
  - Scientific Computations: Scientific Computations (Computing the Strain & Stresses on certain bodies etc..)
  - System Programming: Operating systems etc.. a bit higher than the hardware level.
  - Artificial intelligence
- 21st Century: Retains the above, but also includes:
  - Publishing
  - Process Description and Control
  - PC programming: GUIs and interactions between humans and machine
  - Web-programming (information processing of various types, vue.js and whatever)
  - Mobile Applications
  - Embedded System Programming (Such as aircraft systems or whatever) and are very low level with their interaction with hardware.

1.2.2 Hardware/Software trends

Hardware Trends

Mainframes (Batch processing Environments), Minicomputers, Personal Computers (which are getting faster), supercomputers (massively parallel processing), and Embedded Systems (Which require high reliability).
in the late 1970s, Networking started becoming popular: from 1970s, PCs started supporting LANs, in the 80s appeared ARPANET, and in the late 80s, internet and WWWW appeared with the first browser, netscape.
Moore's Law states that the number of transistors in a dense integrated circuit doubles about every two years, this affects Memory size (RAM), Secondary storage (permanent storage), and processor speeds.
This eventually enabled us to have portable devices with high processing power and high memory in the form of laptops and smartphones.

These software trends led to the development of softwar eengineering methodologies, software designing tools, and eventually new PLs (programming languages).

1.2.3 Other factors

Programming environments, Language Standardization (before, a lot of companies had different standards, which made it so that you might not be able to port your program from one compiler to another, as they differed), internationalization,Theoretical studies, Economics.

1.3 PL generations

- 1st generation: machine language
- 2nd generation: assembly language
- 3rd generation: Original "high-level" PLs
  Examples: C, Pascal, Algol, FORTRAN, Java, C++, Lisp, Python?
- 4th generation: "Special purpose"
  provide a higher level of abstraction of the internal computer hardware details
  E.g database management, web-development, mathematical computation, GUI, shell programming
- 5th generation: constraint-based and logic programming languages and some declarative Ls
  E.g Prolog, Lisp, Haskell.

1.3.1 Machine Language (Example)

Simple machine language for a controller that operates a bulb might include shifting between 1s and 0s to shift between the operational states of the machine.

1.3.2 Assembly Language (Example)

Instructions start in a specific column, end in a specfic column, and are generally more structured than machine language.
This is where hexadecimal numbers start to appear, because you can include 4 bits in a single hex digit.

1.3.3 Evolution of imperative & OOP PLs

FORTRAN IV (1957) was the first PL to have a structured programming paradigm, and it was the first PL to have a compiler.
Ada 83 and 95 were extremely complex and were some of the first languages to be fully operatable.

1.3.4 Fortran IV

GOTO is a reserved keyword in FORTRAN IV, and it is used to jump to a specific line in the program.
the DO-loop was the only high-level instruction in FORTRAN IV.

1.3.5 COBOL (1959-61)

COBOL was introduced by committee, and it was the first PL to be used in business applications.
Its features included separate data description, record data structures, file description/-manipulation, and early Standardization.
Supera Al Akhawayn has an ex-COBOL programmer.
A sample COBOL program is shown in the course slides.
BLOCK Structured language
Compound statements let us treat a series of statements like a single statement.
Blocks are compound statements + Local DATA.

1.3.6 ALGOL 60 (1957-60)

Thsi was the first language carefully defined by "report".
It was block structured, and could handle recursion.

It had explicit data type declarations, scope rules and dynamic lifetimees, relational & boolean expressions, and more.
It is now not used much, but was used back then.
ABSTRACT DATA TYPES
a data structure = ADT + implementation.
LIST ADT:

- Sequences of elements
- Ordering
- Successor (next)
- Predecessor (Previous)

A string and a list are both sequences, and both are considered subtypes of the type SEQUENCE.

1.3.7 Modular Languages

They started distinguishing between the **interface** and the **implementation**.
Interface: the set of operations that can be performed on the data structure.
Implementation: the way the data structure is implemented.

1.3.8 Smalltalk (1976,1980)

This was the first fully object oriented PL.
It was developed by Alan Kay, and it was the first PL to have a graphical user interface.
It had objects and object abstractions (classes)

1.3.9 C++

C++ was built on top of C, and is still widely used, and comparable to Java.
The main advantage of C++ is that it is compatible with C, and you could opt to not make everything be an object.

1.4 Summary

Architecture has affected which language are used and which are not.
You can think of these languages as belonging to different paradigms based on two different criteria:

- The way the program is structured
- The way the data is structured

Chapter 2

Computational models and Paradigms

2.1 Computational Models

2.1.1 Imperatives

Focuses on decomposition into steps and the routines used to modularize. Examples include Pascal and C.

Control Structures	Implementation
Sequence	One statement after another
Selection	Various types of conditional statements
Repetition	Loops

2.1.2 Functional

Computation of values by use of expressions and functions. Works by passing values to functions and returning values from functions. Examples include Lisp and Haskell.

Control Structures	Implementation
Sequence	Function Application
Selection	Various types of conditional statements
Repetition	Recursion

Example: Computing the average of a list of numbers.

2.1.3 Logic

Computation of truth values by use of logical expressions and rules. Examples include Prolog and Datalog.
Example of prolog:

prologfact(1,1).
prologfact(N,F) :-
bla bla bla

2.2 Machine architecture

Basic model is von neumann architecture: Works with a fetch, execture store cycle.
Instructions are low level and change state of machine.
The von Neumann bottleneck: speed of machine limited by the speed of memory-CPU connection.
An example of a functional language is Lisp which was used by Lisp Machines, but they eventually, they did not succeed (see wikepedia for details).
Also, refer to wikepedia for "Timeline of Programming Languages" and "Programming Paradigms".

Chapter 3

Source code Execution

3.1 A compiler is a translator

The compiler takes a source program in a source langauge and transforms it into a target language.
e.g: C programs and Java documents and LaTeX documents are source programs.
Target programs are machine code, Java byte code, PS documents, etc.

3.1.1 Why study compilers?

- TO be more effective users of compilers ... instead of treating it as a black box.
- Apply compiler techniques to typical Software Engineering tasks that require reading input and taking action.
  ex: You have a system where you take reservations for train travel, where the user takes a reservation. You might want to apply some compiler techniques to extract some data and store them into data structures and perform some oeprations on them.
- To see how core CS courses fit together, giving you the knowledge to construct a compiler.
  You might want to see how something behaves in a data structure
- To participate in R&D projects to develop new high level PLs.

3.1.2 The phases of a compiler

Source goes goes from a scanner, which is also called Lexer, and it stands for Lexical Analyzer. It scans words, reserved words, punctuations etc..
It transforms your text file into a list of tokens, which carries information from the source code.
Some information from your file won't be needed.
Those tokens get put into a parser, which does synctactic analysis.

Synctactic Analysis is tasked with checking whether the tokens are in the expected sequence.
And it uses for this purpose: Grammar. A formal description of what the syntax of the language is.
We obtain a Concrete Syntax Tree, this however contains a lot of information that you won't need later on.
It will allow you to check things about the grammar that the parser was not able to check.
This is called the Static Semantics Analyzer, but semantics is not used correctly used here, it has to do with types and "have we ever seen what this is before". instead of the meaning.
This leads to an abstract sytnax tree which is annotated or decorated. This is the front end of the compiler.
The back end of the compiler includes three parts:

- Source code optimizer: This tries to optimize the code by looking at the tree. Some optimizations are done by rearranging the tree, like some variables you computed in some weird places. You can move them to the top of the tree.

- Code generator: We can generate what we call target code: this is either assembly language or machine language, but it could still be something higher level but not as high level as the source code.

- Target code optimizer: This can be some local optimizations contrarily to the previous parts which take care of global optimizations.

Example of a Lexical Analysis for the code in slide 6: The reserved words get turned into tokens that look like the original reserved words. There's a one to one mapping between the reserved indentifiers.
But the variables for example, they get turned into tokens that look like this: `ID` and then the name of the variable.
A floating point constant would be turned into something like this: `FCONST` and then the value of the constant.

Type of errors in a compiler

- **Compilation errors**: " yes you can do that/ no you can't do that".
- **Runtime errors**: ex: you divide by zero.
- **Logical errors**: ex: you have a loop that never terminates.

```
x = 0; f =1; while (x != n)x = x +1;f = f * x;
```

Example of Semantics

3.2 Compilation

Object code = Machine Language written in a format to be understood by the operating system.
The target program in object code has an input and an output, or at least one of them.

3.2.1 From Compilation to execution

1. The code is in different files
    - user-coded files are compiled separately, could be one after the other or in some other order.
    - Libraries are already compiled, these could be like stdio.h for c or PySINDy for python.
2. The **Linker** resolves external references and pulls everything together. This checks for things like the libraries that you are using, and it checks if they are available.
    - Functions in PL libraries
    - functions in other user-coded files

    An example for this is the `input` function in LaTeX.
    This can lead to some problematic errors such as runtime errors.
    Hard-coded paths can cause problems when you move the code to another machine.
3. The linker creates an **Executable file** which is a file that can be executed.
4. The **Loader** loads the executable file into memory.
    Most of the code that is stored in your .o files use what's called relative addresses, they're relative to the rest of the code.
    We have an absolute address which is the address of the first byte of the executable file.
    This address can be computed from the relative addresses.

3.3 What is an interpreter?

The interpretation does the same process as compilation but only for one statement at a time.

If the statement requires some input, it will go fetch some input.

Chapter 4

A very Simple Programming Language

4.1 The first Programming Languages

4.1.1 Machine Languages

These machine languages are based on simple instructions, they weren't necessarily specified in that way in the sense that you wouldn't see the words we would specify in a bit, but you would maybe see a sequence of 0s and 1s to specify one of these instructions:

- Load
- Store
- Add
- Jump

Your instructions will depend on the architecture of processor:

What is the word size of the machine?

A word is a unit of information (code/data) that is exchanged between CPU & Main memory. The 64-bit and 32-bit words are indication of the word size of the machine.

How much memory must be addressed?

Are there registers? How many? Registers are very fast memory locations that are directly accessible by the CPU.

RISC (Reduced Instruction Set Computer) processors have a small number of registers, while CISC (Complex Instruction Set Computer) processors have a large number of registers.

4.1.2 Machine vs. Assembly Languages

Machine languages are sequences of 0s and 1s, while assembly languages are sequences of mnemonics (words) that are mapped to machine instructions.

When programming languages weren't very developped yet, it was extremely difficult to write programs in machine languages, because you had to remember a lot of

mnemonics, optimize CPU and memory usage and so on.

4.1.3 Floating point operations

This is one of the problems that led to the development of subroutines.

4.1.4 Indexing

We used array for scientific computations, but somtimes we needed to map multi dimensional arrays to memory locations, which were one dimensional.
This was first done through address modification, which was very dangerous.

4.1.5 Pseudo-Code Interpreters

Floating point and indexing subroutines allowed programming as if hardware provided operations for them.
This sparked a new idea, which led to creation of Pseudo-Code and an Interpreter.
When talking about Virtual machines we are talking about a machine that has its own data types and operators, it's guided by design principles, so that you can remember easier how to use it.
→ This was part A of the Slides.

4.2 Designing a Simple Language

4.2.1 The first Language

This is not an existing language, it's just a language that we will use to illustrate the concepts of programming languages.
It is a very small machine compared to today's machines, which has 2000 words of 10-digit memory. These numbers are decimal. These digits will have a sign, so they can be positive or negative.
This machine has some basic operations, such as arithmetic and comparison, it also knows how to index arrays, which is more sophisticated than machine languages.
It could also transfer control, which means jumping to a different line of the code.
You also have some simple Input/Output operations, such as reading and writing.

4.2.2 What is a Programing

A list of instructions stored in memory.
An instruction is an opcode + operands. With 1 instruction equal to 1 word.
There are two designs for instructions: both of them have 1 word (10 digits and a sign

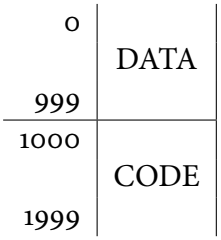
bit)
One has 2 digits opcode, 8 digits for 2 operands, the other has 1 digit for the opcode, with a three operands of 3 digits each. These operands symbolize memory locations.

4.2.3 Things to think about

Do you mix instructions and data in memory?
Sometimes, like we do in modern programming languages, you could format your code in a way that data and code are shuffled randomly in memory. A second option would be to have a separate block for Data and one for Code.
How many operators (opcodes) Operands? How many ? Do you need to represent them all ? What is the size of the operands ?
What do these operands represent? Addresses? Values?

4.2.4 Main Memory Organization

We will only have 2000 words of memory, which we will split into:



4.2.5 Instrucion Design

- **Operators** only need n, n between 10 and 20 operators, so we can use 1 sign and 1 digit. This combo is called an opcode.

- **Operands** 3 operands of 3 digits each.
  These will all be memory addresses.

4.2.6 Design Principle

Splitting memory into 2 parts and using the convention that certain operands will olny refer to data memory and others only to program memory memory embodies an important principle.

Impossible Error Principle Making it impossible to make errors is better than making it easy to detect errors.
With this design, coding `c = a + b` required one instruction:

$$\texttt{ADD}\ \ a_{mem}\ \ b_{mem}\ \ c_{mem}$$

4.2.7 Summary: Introducition Syntax and Semantics

Syntax

+/−	OP	OPD1	OPD2	DEST

Semantics:
dest ← opd1 op od2
→ This was part B of the Slides.

4.3 How to code operations?

4.3.1 Regularity Principle

Regular rules, without exceptions, are easier to learn, use, design and debug.

4.3.2 Orthogonality and Symmetry

Operation	op code	+	-
add	1	+	-
multiply	2	*	/
quadratic	3	x^2	$\sqrt{x}$

There is some regularity in this, on one side, we have a clear relationship between the positive and negative signs, and the further we gown down the opcode lists, the further the complexity increases.

Therefore, we will map the direction to the sign, and the complexity to the # code. This has to do with the word orthogonality.

4.3.3 Pushing Symmetry and Orthogonality too far

We might run into exceptions, which come with the introduction fo irregular rules.
The orthogonality is advantageous if:

$$m + n + e < m * n + e \tag{4.3.1}$$

with e being the number of exceptions.

Example from before $m = 10$ and $n = 20$ and $e = 1$
$m + n + e = 31$ and $m * n + e = 201$
Thus we can see that the orthogonality is advantageous.

4.3.4 Rrepresenting Control Flow

We might introduce equality and inequality operators, for example:

Equality test + branch	4	if = goto	if ≠ goto
Inequality test + branch	5	if ≥ goto	if < goto

Just greater than or equal and less than are enough because you can obtain the other 2 by switching the two operands.
To compare to 0, we can store the value 0 in memory location 000.

4.3.5 Simple Assignment/Moving Data

Do we really need a special operator?
We can use the `ADD` operator to move data.
But this is expensive as it require a couple of instructions.

Chapter 5

Lexical Analysis

5.1 Reminder about compilation steps

1. **Scanner/Lexer**: This takes a Source program and breaks it into tokens.
2. **Parser**: This takes a stream of tokens and builds a parse tree.
3. **Semantic Analyzer**: Takes cares of the annotations and type checking. This generated an annotated tree.

From here on this low level optimization

---

4. **Source code optimizer**: Converts the Annotated tree into intermediate code.
5. **Code Generator**: Generates the target code.
6. **Target Code Optimizer**: Optimizes the target code.

It is important to note that a Lexical Analyzer is a synonym of a Lexer and a Scanner. The terms are used interchangeably.

5.2 Lexical Analysis

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. The tokens are the smallest meaningful units of a programming language. For example, in C, the tokens are identifiers, keywords, operators, and so on. The lexical analyzer is also known as a scanner or a lexer.
This removes the content-free characters such as spaces, tabs, and newlines. It also removes comments.
It detects lexical errors, such as badly-formed literals, and illegal characters.
This is a pattern matching process: It acts as front-eend for the parser, and is an implementation of a finite state automaton.
A substring of the source program that belongs together is called a lexeme, and a lexeme is matched against a pattern, which is associated with a lexical category called a **token**.

You can define a lexical analyzer via a list of pairs:

- A regular Expression: describes a token pattern that is matched against the input to extract lexemes.
- An action: a piece of code, parameterized by the matching lexeme, that returns a (token, attribute) pair.

5.3 Regular Expressions

These are a language with its own syntax and semantics. They are used to describe patterns in strings.
Certain characters and character combinations denote specific operations, impose certain constraints, or modify default operator precedence. **Meta languages**: They describe a particular family of languages called the **regular languages**.

5.3.1 Important Symbols

The empty set: $\varnothing$
The empty string: ϵ
Alternation: |
Repetition: $*$: 0 or more times, +: 1 or more times, ?: 0 or 1 times
Optionality: ?: a? describes the set of strings that are either a or ϵ
A kleene cross: +: a+ describes the set of strings that are either a or aa or aaa or ...
Wildcard: .: matches any single character
Classes: $[a-z]$
Escape: \

5.3.2 Precedence

The precedence of the operators is as follows:

1. $*$, +, ?, *, $^+$, $^?$, $^-$
2. Concatenation
3. Alternation

5.4 Finite State Automata

A finite state automaton (FSA) is a machine that has a finite number of states and a finite number of inputs.
It is a model of computation that can be used to recognize languages.
It is a 5-tuple: $M = (Q, \Sigma, \delta, q_0, F)$
Where:

- Q is a finite set of states.
- Σ is a finite set of input symbols.
- $\delta : Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states. (accepting states)

5.4.1 Deterministic Finite State Automata

A deterministic finite state automaton (DFA) is a finite state automaton that has the following properties:

1. For each state $q \in Q$ and each input symbol $a \in \Sigma$, there is at most one transition from q on a.
2. The transition function δ is total: for each state $q \in Q$ and each input symbol $a \in \Sigma$, there is a transition from q on a.

5.4.2 Non-Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is a finite state automaton that is not necessarily deterministic.
an NFA accepts a string when there is a computation of the NFA that accepts the string.
There is a **Computation** when all the input is consumed and the automaton is in an accepting state.
An NFA rejects a string when there is no computation of the NFA that accepts the string.

Empty transitions are allowed in an NFA, but not in DFAs.

5.4.3 DFA vs NFA

- NFAs are usually smaller but take more time to process, since they have to backtrack if they tak the wrong path, or even hang.
- Every NFA can be transformed into a DFA, but the resulting DFA may be much larger.
- Every NFA with lambda transitions can be transformed into an equivalent NFA without lambda transitions.

5.5 Lexical Analyzer Implementation

You might design a finiet state automaton that d escribes the expected input patterns and writes a program that implements the automaton.
This is a tedious process, and it is error-prone.

You might design a FSA that describes the expected input patterns and hand-construct a table-driven implementation of the FSA.

You might write a formal description of he expected input patterns using REs and use a software tool that constructs table-driven lexical analyzers from such a description.

5.6 State Diagram Design

This is one way of representing an FSA.
A naïve state diagram would have a transition from every state on every character in the source language. But it would be very large!
In many cases, transitions can be combined to simplify the state diagram:

- When recognizing an identifier, all uppercase and lowercase letters are equivalent. So, you can combine all the transitions on the letters into a single transition on a class of letters.
- When recognizing a number, all digits are equivalent. So, you can combine all the transitions on the digits into a single transition on a class of digits.

an example of this code in C would look like:

```
int LexAnalyzer() {
    getChar();
    if (isLetter(nextChar)) {
        addChar();
        getChar();
        while (isLetter(nextChar) || isDigit(nextChar)) {
            addChar();
            getChar();
        }
        return lookup(lexeme);
    }
    else if (isDigit(nextChar)) {
        addChar();
        getChar();
        while (isDigit(nextChar)) {
            addChar();
            getChar();
        }
        return INT_LIT;
```

```
20      }
21  }
```

You could try to match all reserved words explicitly in the FSA, but this would be tedious and error-prone.
It is better to treat all IDs the same but look them up in the **Symbol Table** after the token is created to identify which are reserved words.

A lexical analyzer using token parsing with FSA logic implementation algorithm would look like:

```
state =S // S is the start state
repeat {
    k = next character from the input
    if k == EOF                     // the end of input
        if state is a final state
            then accept
            else reject
    state = transition(state, k)    // transition to a new state
    if state == empty
        then reject                 // got stuck
    }
```

5.7 On Lexical Errors

Lexical errors are not syntax errors, they occur only if a character not recognized by a programming language.
This usually means that you have a malformed lexeme **According to the rules of the language**.

5.8 Practice Problems

This section will be added later on.

Chapter 6

Language Description & Analysis at the Syntactic Level:

6.1 Regular and Context-Free Languages

6.1.1 Regular Languages

A language L is regular if there is an automaton M such that $L(M) = L$. All languages accepted by FAs form the family of regular languages.
There exist automata that accept these simple languages:

- $\{ abbab \}$
- $\{ \lambda, ab, abba \}$

and even these languages:

- All strings with prefix ab
- All strings without the substring 001

We use Regular expressions for description, Finite Automata for Implementation and Regular Languages for Analysis.

6.1.2 Analysis vs. Generation

Analysis is a stronger form of acceptance. It is the ability to determine whether a string belongs to a language or not.
Analysis extracts some information from its input and provides it in the ouput in a transformed form, e.g: Lexical analysis transforms source text into a token stream.

A description of $L(M)$ should describe exactly all the strings in $L(M)$ and no other strings.
Generation uses the formal description (grammar) to output examples of strings belonging to language L.

A description can both **Overgenerate** and **Undergenerate**.

Overgeneration is a description that generates strings that are not in the language. Undergeneration is a description that does not generate all the strings in the language.

6.1.3 Non-Regular Languages

A language L is non-regular if there is no automaton M such that $L(M) = L$.
An example is the following language: $\{ a^n b^n \mid n \geq 0 \}$
This language is not regular because it is not accepted by any FA.
This is because there is no way to remember the number of number of as that have been generated, and then generate the same number of bs.
you would need a stack to remember the number of as generated, and then generate the same number of bs. This woudl be done through a **Pushdown Automaton**.
We can classify languages into 4 sections:

- Regular
- Context-Free
- Context-Sensitive
- Recursively Enumerable

The generative capacity of languages increases as we move from top to bottom.
context-free grammar is used to describe context-free languages, and a parser is used to implement it.

Grammar is a formal system that provides a generate finite description of L using a set of rules.
A grammar is a 4-tuple $G = (V, T, P, S)$ where:

- V is a finite set of **non-terminal symbols**
- T is a finite set of **terminal symbols**
- P is a finite set of **production rules**
- S is the **start symbol**