08176074762, 08089908927

Organization of Programming Languages (CSC 411)

A programming language is an artificial language designed for expressing algorithms on a computer.

A programming language is a **computer language** that is used by **programmers (developers) to communicate with computers**. It is a set of instructions written in any specific language (C, C++, Java, Python) to perform a specific task.

A programming language is mainly used to **develop desktop applications, websites, and mobile applications**.

The benefits of programming languages as a course

REASONS FOR STUDYING CONCEPTS OF PROGRAMMING LANGUAGES

1. Increased ability to express ideas/algorithms

In Natural language, the depth at which people think is influenced by the expressive power of the language they use. In programming language, the complexity of the algorithms that people Implement is influenced by the set of constructs available in the programming language

2. Improved background for choosing appropriate Languages

Many programmers use the language with which they are most familiar, even though poorly suited for their new project. It is ideal to use the most appropriate language.

3. Increased ability to learn new languages

For instance, knowing the concept s of object oriented programming OOP makes learning Java significantly easier and also, knowing the grammar of one's native language makes it easier to learn another language.

4. Better Understanding of Significance of implementation

5. Better use of languages that are already known

6. The overall advancement of computing

APPLICATION DOMAINS

1. Scientific Applications
2. Data processing Applications
3. Text processing Applications
4. Artificial intelligence Applications
5. Systems Programming Applications
6. Web software

SCIENTIFIC APPLICATIONS can be characterized as those which predominantly manipulate numbers and arrays of numbers, using mathematical and statistical principles as a basis for the algorithms. These algorithms encompass such problem as statistical significance test, linear programming, regression analysis and numerical

approximations for the solution of differential and integral equations. FORTRAN, Pascal, Meth lab are programming languages that can be used here.

DATA PROCESSING APPLICATIONS can be characterized as those programming problems whose predominant interest is in the creation, maintenance, extraction and summarization of data in records and files.
COBOL is a programming language that can be used for data processing applications.

TEXT PROCESSING APPLICATIONS are characterized as those whose principal activity involves the manipulation of natural language text, rather than numbers as their data. SNOBOL and C language have strong text processing capabilities

ARTIFICIAL INTELLIGENCE APPLICATIONS are characterized as those programs which are designed principally to emulate intelligent behavior. They include game playing algorithms such as chess, natural language understanding programs, computer vision, robotics and expert systems. LISP has been the predominant AI programming language, and also PROLOG usingthe principle of "Logic programming" Lately AI applications are written in Java, c++ and python.

SYSTEMS PROGRAMMING APPLICATIONS involve developing those programs that interface the c omputersystem (the hardware) with the programmer and the operator. These programs include compilers, assembles, interpreters, input-output routines, program management facilities and schedules for utilizing and serving the various resources that comprise the system. Ada and Modula – 2 are examples of programming languages used here. Also is C.

WEB SOFTWARE
Edectio collection of languages which include:
- Markup (e.g. XHTML)
- Scripting for dynamic content under which we have the o Client side, using scripts embedded in the XHTML documents e.g. Javascript, PHP o Server side, using the commonGateway interface e.g. JSP, ASP, PHP
- General- purpose, executed on the web server through cGI e.g. Java, C++.

CRITERIA FOR LANGUAGE EVALUATION AND COMPARISION
1. Expressivity
2. Well –Definedness
3. Data types and structures
4. Modularity
5. Input-Output facilities
6. Portability
7. Efficiency
8. Pedagogy
9. Generality

Expressivity means the ability of a language to clearly reflect the meaning intended by the algorithm designer (the programmer). Thus an "expressive" language permits an utterance to be compactly stated, and encourages the use of statement forms associated with structured programming (usually "while "loops and "if – then – else"statements).

By "well-definiteness", we mean that the language's syntax and semantics are free of ambiguity, are internally consistent and complete. Thus the implementer of a well-definedlanguage should have, within its definition a complete specification of all thelanguage's expressive forms and their meanings. The programmer, by the same virtue should be able to predict exactly the behavior of each expression before it is actuallyexecuted.

By "Data types and Structures", we mean the ability of a language to support a variety of data values (integers, real, strings, pointers etc.) and non-elementary collect ions of these.

Modularity has two aspects: the language's support for sub-programming and the language's extensibility in the sense of allowing programmer – defined operators and data types. By sub programming, we mean the ability to define independent procedures and functions (subprograms), and communicate via parameters or global variables with the invoking program.

In evaluating a language's "Input-Output facilities" we are looking at its support for sequential, indexed, and random access files, as well as its support for database and information retrieval functions.

A language which has **"portability"** is one which is implemented on a variety of computers. That is, its design is relatively "machine – independent". Languages which are well- defined tend to be more portable than others.

An **"efficient"** language is one which permits fast compilation and execution on the machines where it is implemented. Traditionally, FORTRAN and COBOL have been relatively efficient languages in their respective application areas.

Some languages have better **"pedagogy"** than others. That is, they are intrinsically easier to teach and to learn, they have better textbooks; they are implemented in a better program development environment, they are widely known and used by the best programmers in an application area.

Generality: Means that a language is useful in a wide range of programming applications. For instance, APL has been used in mathematical applications involving matrix algebra and in business applications as well.

INFLUENCES ON LANGUAGE DESIGN

1. Computer Architecture: Languages are developed around the prevalent computer architecture, known as the Von Neumann architecture (the most prevalent computer architecture).The connection speed between a computer's memory and its processor determines the speed of that computer. Program instructions often can be executed much faster than the speed of the connection; the connection speed thus, results in a bottleneck (Von Neumann bottleneck). It is the primary limiting factor in the speed of computers.

2. **Programming Methodologies:** New software development methodologies (e.g. object Oriented Software Development) led to new paradigms in programming and by extension, to new programming languages.

LANGUAGE PARADIGMS (Developments in Programming Methodology)

1. Imperative

This is designed around the Von Neumann architecture. Computation is performed through statements that change a program's state. Central features are variables,assignment statements and ileration, sequency of commands, explicit state update via assignment. Examples of such languages are Fortran, Algol, Pascal, e/c++, Java, Perl,
Javascript, Visual BASIC.NET.

2. Functional

Here, the main means of making computations is by applying functions to parameters.
Examples are LISP, Scheme, ML, Haskell. It may also include OO (Object Oriented) concepts.

3. Logic

This is Rule-based (rules are specified in no particular order). Computations here are made through a logical inference process. Examples are PROLOG and CLIPS. This may also include OO concepts.

TRADE-OFFS IN LANGUAGE DESIGN

1. Reliability Vs. Cost of Execution: For example, Java demands that all references to array elements be checked for proper indexing, which leads to increased execution costs.
2. Readability vs. Writability: - APL provides many powerful operators land a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.
3. Writability (Flexibility) vs. reliability: The pointers in c++ for instance are powerful and very flexible but are unreliable.

IMPLEMENTATION METHODS

1. Compilation – Programs are translated into machine Language & System calls
2. Interpretation – Programs are interpreted by another program (an interpreter)
3. Hybrid – Programs translated into an intermediate language for easy interpretation
4. Just –in-time – Hybrid implementation, then compile sub programs code the first time they are called.
5. Pure Interptretation

COMPILATION

- Translated high level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases

*Lexical analysis converts characters in the source program into
*lexical units (e.g. identifiers, operators, keywords).
*Syntactic analysis: transforms lexical units into parse trees whichrepresent *he syntactic structure of the program.
*Semantics analysis check for errors hard to detect during syntacticanalysis; generate intermediate code.
*Code generation – Machine code is generated

- **INTERPRETATION**
- Easier implementation of programs (run-time errors can easily andimmediately be displayed).
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more memory space and is now rare3 for traditional highlevellanguages.
- Significant comeback with some Web scripting languages like PHP and JavaScript.
- Interpreters usually implement as a read-eval-print loop:
*Read expression in the input language (usually translating it insome internal form)
*Evaluates the internal forms of the expression
*Print the result of the evaluation
*Loops and reads the next input expression until exit
- Interpreters act as a virtual machine for the source language:
*Fetch execute cycle replaced by the read-eval-print loop
*Usually has a core component, called the interpreter "run-time"that is a compile program running on the native machine.

HYBRID IMPLEMENTAITON
- This involves a compromise between compilers and pure interpreters. A high level program is translated to an intermediate language that allows easy interpretation.
- Hybrid implementation is faster than pure interpretation. Examples of the implementation occur in Perl and Java.
*Perl programs are partially compiled to detect errors beforeinterpretation.
*Initial implementat6ions of Java were hybrid. The intermediateform, byte code, provides portability to any machine that has a bytecode interpreter and a run time system (together, these are called Java Virtual Machine).

JUST-IN-TIME IMPLEMENTATION
This implementation initially translates programs to an intermediate language then compile the intermediate language of the subprograms into machine code when they are called.
- Machine code version is kept for subsequent calls. Just-in-time systems are widely used for Java programs. Also .NET languages are implemented with a JIT system.

Study Questions:
1. Why is it useful for a programmer to have some background in language design, even though he or she may never actually design a programming language?

2. What does it mean for a program to be reliable?
3. What is: Aliasing?; What is exceptional handling?
4. Why is readability important in writability?
5. What are the three fundamental features of an object-oriented language?
6. What role does symbol table play in compiler?
7. What does a linker do?
8. What are the advantages in implementing a language with pure interpreter?
9. What are the three general methods of implementing a programming language?
10. Which produces faster program execution, a compiler or a pure interpreter and how?

A BRIEF HISTORY OF PROGRAMMING LANGUAGES.

Assembly languages
IBM 704 and Fortran–FORmulaTRANslation
LISP –LISt Processing
ALGOL 60 – International Algorithmic language
Simul 67 – First object oriented language
Ada – history's largest design effort
C++ - Combining Imperaive and Object – Oriented Features
Java – An Imperative – Based Object – Oriented language
Prolog– Logic Programming

ALGOL

ALGOL 68 (short for **ALGO**rithmic**L**anguage 19**68**) is an imperative computer programming language that was conceived as a successor to the ALGOL 60 programming language, designed with the goal of a much wider scope of application and more rigorously defined syntax and semantics. ALGOL 68 features include expression-based syntax, user-declared types and structures/tagged-unions, a reference model of variables and reference parameters, string and array and matrix slicing and also concurrency.

ALGOL 68 was designed by IFIP Working Group 2.1. On December 20, 1968 the language was formally adopted by Working Group 2.1 and subsequently approved for publication by the General Assembly of IFIP.

ALGOL 68 was defined using a two-level grammar formalism invented by Adriaanvan Wijngaarden. Van Wijngaarden grammars use a context-free grammar to generate an infinite set of productions that will recognize a particular ALGOL 68 program; notably, they are able to express the kind of requirements that in many other programming language standards are labelled.

Notable language elements

Bold symbols and reserved words

There are 61 such reserved words (some with "brief symbol" equivalents) in the standard sub-language:**mode**, **op**, **prio**, **proc**, **flex**, **heap**, **loc**, **long**, **ref**, **short**, **bits**, **bool**, **bytes**, **char**, **compl**, **int**, **real**, **sema**, **string**, **void**, **channel**, **file**, **format**, **struct**, **union**, **of**, **at** "@", **is** ":=:", **isnt**":/=:", ":≠:", **true**, **false**, **empty**, **nil** "○", **skip** "~", **co comment** "¢",

pr, **pragmat**, **case in ouse*in* outesac**"(~ | ~ |: ~ | ~ | ~)", **for from to by while do od**, **if then elif*then* else fi** "(~ | ~ |: ~ | ~ | ~)", **par begin end** "(~)", ***go to***, **goto**, **exit** ".". **mode: Declarations**
The basic data types (called **mode**s in ALGOL 68 parlance) are **real**, **int**, **compl**(complex number), **bool**, **char**, **bits** and **bytes**. For example:
intn = 2;
con is a fixed constant of 2.**co**
intm := 3;
com is a newly created local variable whose value is initially set to 3.
This is short for **ref int**m = **locint**:= 3; **co**
realavogadro = 6.0221415□23; **co** Avogadro's number **co**
longlong real pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510;
complsquare root of minus one = 0 ⊥1
However, the declaration **real** x; is just syntactic sugar for **ref real** x = **loc real**;. That is, x is really the *constant identifier* for a *reference to* a newly generated local **real** variable.

Special characters
Most of Algol's "special" characters (×, ÷, ≤, ≥, ≠, ¬, ∨, ∧, □, →, ↓, ↑, □, ⌊, ⌈, ⌊, ⌈, ○, ⊥and ¢) can be found on the IBM 2741 keyboard with the APL "golf-ball" print head inserted, these became available in the mid 1960s while ALGOL 68 was being drafted. These characters are also part of the unicode standard and most of them are available in several popular fonts.

Transput: Input and output
Transputis the term used to refer to ALGOL 68's input and output facilities. There are pre-defined procedures for unformatted, formatted and binary transput. Files and other transput devices are handled in a consistent and machine-independent manner. The following example prints out some unformatted output to the standard output device: print ((newpage, "Title", newline, "Value of i is ", i, "and x[i] is ", x[i], newline)) Note the predefined procedures newpage and newline passed as arguments.

C++
C++ (pronounced "see plus plus") is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is regarded as an intermediatelevellanguage, as it comprises a combination of both high-level and low-level language features. It was developed by BjarneStroustrup starting in 1979 at Bell Labs as an enhancement to the *C* language and originally named *C* with Classes. It was renamed *C++* in 1983.
C++ is one of the most popular programming languages and its application domains include systems software (such as Microsoft Windows), application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games.

C++ is sometimes called a hybrid language; it is possible to write object oriented or procedural code in the same program in C++. This has caused some concern that some C++ programmers are still writing procedural code, but are under the impression that it is object orientated, simply because they are using C++. Often it is an amalgamation of the two. This usually causes most problems when the code is revisited or the task is taken over by another coder.
C++ continues to be used and is one of the preferred programming languages to develop professional applications.

Language features
C++ inherits most of C's syntax. The following is BjarneStroustrup's version of the Hello world program that uses the C++ standard library stream facility to write a message to standard output:

```
#include <iostream>
int main()
{
std::cout<< "Hello, world!\n";
}
```

Within functions that define a non-void return type, failure to return a value before control reaches the end of the function results in undefined behaviour (compilers typically provide the means to issue a diagnostic in such a case). The sole exception to this rule is the main function, which implicitly returns a value of zero.

Operators and operator overloading
C++ provides more than 35 operators, covering basic arithmetic, bit manipulation, indirection, comparisons, logical operations and others. Almost all operators can be overloaded for user-defined types, with a few notable exceptions such as member access (. and .*). The rich set of overloadable operators is central to using C++ as a domain-specific language. The overloadable operators are also an essential part of many advanced C++ programming techniques, such as smart pointers. Overloading an operator does not change the precedence of calculations involving the operator, nor does it change the number of operands that the operator uses (any operand may however be ignored by the operator, though it will be evaluated prior to execution).
Overloaded "&&" and "||" operators lose their short-circuit evaluation property.

C#
During the development of the .NET Framework, the class libraries were originally written using a managed code compiler system called Simple Managed C (SMC). In January 1999, Anders Hejlsberg formed a team to build a new language at the time called Cool, which stood for "C-like Object Oriented Language". Microsoft had considered keeping the name "Cool" as the final name of the language, but chose not to do so for trademark reasons. By the time the .NET project was publicly announced at the July 2000 Professional Developers Conference, the language had been renamed C#, and the class libraries and ASP.NET runtime had been ported to C#.
Some notable distinguishing features of C# are:

1. It has no global variables or functions. All methods and members must be declared within classes. Static members of public classes can substitute for global variables and functions.
2. Local variables cannot shadow variables of the enclosing block, unlike C and C++. Variable shadowing is often considered confusing by C++ texts.
3. C# supports a strict Boolean data type, bool. Statements that take conditions, such as while and if, require an expression of a type that implements the true operator, such as the boolean type. While C++ also has a boolean type, it can be freely converted to and from integers, and expressions such as if(a) require only that a is convertible to bool, allowing a to be an int, or a pointer. C# disallows this "integer meaning true or false" approach, on the grounds that forcing programmers to use expressions that return exactly bool can preventcertain types of common programming mistakes in C or C++ such as if (a = b) (use of assignment = instead of equality ==).
4. In addition to the try...catch construct to handle exceptions, C# has a try...finally construct to guarantee execution of the code in the finally block.
5. C# currently (as of version 4.0) has 77 reserved words.
6. Multiple inheritances are not supported, although a class can implement any number of interfaces. This was a design decision by the language's lead architect to avoid complication and simplify architectural requirements throughout CLI.

Common Type System (CTS)
C# has a unified type system. This unified type system is called Common Type System (CTS). A unified type system implies that all types, including primitives such as integers, are subclasses of the System.Object class. For example, every type inherits a ToString() method. For performance reasons, primitive types (and value types in general) are internally allocated on the stack.

Categories of data types
CTS separate data types into two categories:
1. **Value types:** they are plain aggregations of data. Instances of value types do not have referential identity nor a referential comparison semantics - equality and inequality comparisons for value types compare the actual data values within the instances, unless the corresponding operators are overloaded. Value types are derived from System. ValueType, always have a default value, and can always be created and copied. Some other limitations on value types are that they cannot derive from each other (but can implement interfaces) and cannot have an explicit default (parameterless) constructor. Examples of value types are all primitive types, such as int (a signed 32-bit integer), float (a 32-bit IEEE floating-point number), char (a 16-bit Unicode code unit), and System.DateTime (identifies a specific point in time with nanosecond precision). Other examples are enum(enumerations) and struct (user defined structures).

2. **Reference types:** they have the notion of referential identity - each instance of a reference type is inherently distinct from every other instance, even if the data within both instances is the same. This is reflected in default equality and inequality

comparisons for reference types, which test for referential rather than structural equality, unless the corresponding operators are overloaded (such as the case for System.String). In general, it is not always possible to create an instance of a reference type, nor to copy an existing instance, or perform a value comparison on two existing instances, though specific reference types can provide such services by exposing a public constructor or implementing a corresponding interface (such as ICloneable or IComparable). Examples of reference types are object (the ultimate base class for all other C# classes), System.String (a string of Unicode characters), and System.Array (a base class for all C# arrays). Both type categories are extensible with user-defined types.

Study Questions

1. Make an educated guess as to the most common syntax error in Lisp programs.
2. Describe in detail the three most important reasons, in your opinion, why ALGOL 60 did not become a very used language.
3. Do you think language design committee is a good idea? Support your opinion.
4. Give a brief general description of a markup/programming hybrid language
5. Why in your opinion, do new scripting languages appear more frequently than new compiled languages?
6. Write a program to implement N factorial in; Machine Language, Assembly Language, Scripting Language and any other five high level languages.

Objective:

The objective of the week lecture is for the student to be able to understand Generative Grammars, Lexical Analysis, Syntactic Analysis and Finite Automata.

Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

SYNTAX

A programming language's surface form is known as its syntax. The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure).

Semantics

The term semantics refers to the meaning of languages, as opposed to their form (syntax). Defines the meaning of a given grammar.

Semantics Xteristics

Various approaches: Operational Semantics „ Denotational Semantics „ Axiomatic Semantics
Attributes and Bindings
Scope (static vs dynamic)
The symbol table (nested or stacked)

Dealing with dynamic memory, aliases, and garbage collection.

Static semantics
The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding an integer to a function name), or that subroutine calls have the appropriate number and type of arguments, can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic semantics
Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The dynamic semantics (also known as execution semantics) of a language defines how and when the various constructs of a language should produce program behaviour. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Grammar
Below is a simple grammar, based on Lisp:
expression ::= atom | list
atom ::= number | symbol
number ::= [+-]?['0'-'9']+
symbol ::= ['A'-'Z''a'-'z'].*
list ::= '(' expression* ')'
This grammar specifies the following:

- An expression is either an atom or a list;
- An atom is either a number or a symbol;
- A number is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- A symbol is a letter followed by zero or more of any characters (excluding whitespace); and
- A list is a matched pair of parentheses, with zero or more expressions inside it.

Syntactic ambiguity
Syntactic ambiguity is a property of sentences which may be reasonably interpreted in more than one way, or reasonably interpreted to mean more than one thing.
Ambiguity may or may not involve one word having two parts of speech or homonyms. Syntactic ambiguity arises not from the range of meanings of single words, but from the relationship between the words and clauses of a sentence, and the sentence structure implied thereby. When a reader can reasonably interpret the same sentence as having more than one possible structure, the text is equivocal and meets the definition of syntactic ambiguity.

Operator Precedence

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before other parts. Operations within parentheses are always performed before those outside.
Within parentheses, however, normal operator precedence is maintained.
When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence:

Arithmetic	Comparison	Logical
Exponentiation (**^**)	Equality (**=**)	**Not**
Negation (**-**)	Inequality (**<>**)	**And**
Multiplication and division (*****, **/**)	Less than (**<**)	**Or**
Integer division (****)	Greater than (**>**)	**Xor**
Modulus arithmetic (**Mod**)	Less than or equal to (**<=**)	**Eqv**
Addition and subtraction (**+**, **-**)	Greater than or equal to (**>=**)	**Imp**
String concatenation (**&**)		**Is&**

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.
The string concatenation operator (**&**) is not an arithmetic operator, but in precedence it does fall after all arithmetic operators and before all comparison operators. The **Is** operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.

Lexical structures, Syntax, parse trees

Lexical Structure of Languages: Define the tokens in a language (the valid characters, keywords, symbols, …)
Grammars and BNF: Defines the structure of a language (the contextual arrangement of the tokens) through a set of formal rules.
Parse Trees: Is a consequence of BNF. The grammar can be represented as a tree, which in turn can serve as a parse tree to check for grammar correctness.

Parsing

In linguistics, parsing is the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Parsing can also be used as a linguistic term, especially in reference to how phrases are divided up in garden path sentences.

Parser

In computing, a parser is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. The parser often uses a separate lexical analyser to create tokens from the sequence of input characters. Parsers may be programmed by hand or may be (semi-)automatically generated (in some programming languages) by a tool.

Overview of process

Types of parser

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar. This can be done in essentially two ways:

· Top-down parsing- Top-down parsing can be viewed as an attempt to find leftmost derivations of an input-stream by searching for parse trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules. Examples includes: Recursive descent parser, LL parser (Left-to-right, Leftmost derivation), and so on.

· Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers. Another term used for this type of parser is Shift-Reduce parsing.

Study questions:
1. Define syntax and semantic
2. What is the primary use of attribute grammars?
3. Define a left recursive grammar
4. The two mathematical models of Language description are generation and recognition. Describe how each can define the syntax of a programming language
5. Write EBNF description for the following:
A Java class definition header statements
A Java method call statement
A C switch statement
A C Union definition
C float literals
6. Prove that the following grammar is ambiguous:
<S> -><A>
<A> -><A> + <A>|<id>
<id> ->a|b|c
7. Which of the following sentences are in the language generated by this grammar
Baab, bbbab, bbaaaaa, bbaab

Attributes of a Good Language

• Cost of use

• Program execution (run time), program translation,

program creation, and program maintenance

• Portability of programs

• Develop on one computer system, run on another

• Programming environment

• External support for the language

• Libraries, documentation, community, IDEs

Clarity, simplicity, and unity

• Provides both a framework for thinking about algorithms

and a means of expressing those algorithms

• Orthogonality

• Every combination of features is meaningful

• Features work independently

• Naturalness for the application

• Program structure reflects the logical structure of

Algorithm

Definition of programming languages Definition: A programming language is a notational system for describing computation in machinereadable and human-readable form.

We differentiated between four programming language paradigms. „ Imperative programming; „ Functional programming; „ Logic programming. „ Object-oriented programming;

SORT

Reasons for studying concepts of programming languages
1. Increased capacity to express ideas.
2. Improved background for choosing appropriate languages.
3. Increased ability to learn new languages.
4. Better understanding of the significance of implementation.
5. Better use of languages that are already known.
6. Overall advancement of computing.

Programming Domains
1. Scientific applications: large numbers of floating point calculations; Fortran
2. Business applications: produce reports, use decimal numbers and characters; COBOL
3. Artificial intelligence: symbols rather than numbers manipulated, linked-lists; LISP
4. Systems programming: need efficiency because of continuous use; C
5. Web software: Eclectic collection of languages

Language Evaluation Criteria
Readability, Writability, Reliability, and Cost (although cost is not in the table)

Language Evaluation Criteria Characteristics
1. Simplicity: Read., Writ., and Rel.
2. Orthogonality: Read., Writ., and Rel.
3. Data Types: Read., Writ., and Rel.
4. Syntax Design: Read., Writ., and Rel.
5. Support for abstraction: Writ., and Rel.
6. Expressivity: Writ., and Rel.
7. Type checking: Rel.
8. Exception handling: Rel.
9. Restricted aliasing: Rel.

Language Categories
1. Imperative: central features - variables, assignment statements, iteration; includes OOP, scripting languages, and visual languages
2. Functional: main means of making computations is by applying functions to given parameters; lambda calculus
3. Logic: rule-based, first order logic, boolean expressions; Prolog
4. Markup/programming hybrid
Implementation Methods

1. Compilation: Programs are translated to machine language; includes JIT systems; Use: Large commercial applications

2. Pure Interpretation: Programs are interpreted by another program known as an interpreter; Use: Small programs or when efficiency is not an issue.
3. Hybrid Implementation Systems: A compromise between compilers and pure interpreters; Use: small and medium systems where efficiency is not the first concern

Phases of the Compilation Process
1. Lexical analysis: converts characters in the source program into lexical units.
2. Syntax analysis: transforms lexical units into parse trees which represent the syntactic structure of the program.
3. Semantics analysis: generate intermediate code
4. Code generation: machine code is generated.
Pure Interpretation
No translation, easier implementation, slower execution, more memory needed
JIT Implementation Systems
1. Translate programs to intermediate language
2. Compile intermediate language into machine code when subprograms are called.
3. Machine code version is kept for subsequent calls.
Language
Characterized by the set of all valid statements in that language
Syntax
Formal method to describe how to determine a statement's set membership in a language
Semantics
The meaning of the expressions, statements, and program units
Lexeme
Lowest level syntactic unit of a language
Token
Category of lexemes
Language Recognizers
A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
Ex: syntax analysis part of a compiler
Generators
A device that generates sentences of a language
One can determine if the syntax of a sentence is correct by comparing it to the structure of the generator.
Formal mechanisms are called grammars.
Context-Free Grammar
Define a class of languages called context-free languages.
G = (S,N,T,P), where:
- G: Grammar
- S: Start symbol
- N: Set of Non-Terminals
- T: Set of Terminals
- P: Set of Production Rules
Backus-Naur Form
Abstractions are used to represent classes of syntactic structures
A rule has a LHS (non-terminal) and a RHS (string of terminals and/or non-terminals)
Start symbol: special element of nonterminals
Sentential Form
Every string of symbols in a derivation
Sentence
Sentential form that has only terminal symbols
Leftmost Derivation
The leftmost nonterminal in each sentential form is the one that is expanded.
Operator Precendence
The lower operation is done first.
To change operator precedence, swap the two operations.
Unambiguous Grammar for if-then-else
<if_stmt>-->if<logic_expr> then <stmt>
if<logic_expr> then <stmt> else <stmt>
EBNF
- Optional parts in brackets []
- Alternative parts of RHSs are placed in parentheses and separated by vertical bars (|)
- Repetitions (0 or more) are placed inside braces {}
Static Semantics
Has to do with the legal forms of programs (syntax rather than semantics)
Dynamic Semantics
Meaning of expressions, statements, and program units

Approaches to Dynamic Semantics
Operational
Denotational
Axiomatic
Operational Semantics
- Describe meaning of a statement or program by specifying the effects of running it on a machine.
Denotational Semantics
- Most rigorous and widely known formal method for describing the meaning of programs.
- Programming language constructs are mapped to mathematical objects.
- Does not model the step-by-step computational processing of programs.
- Uses state of program to describe meaining
Axiomatic Semantics
- Based on mathematical logic
- Specifies what can be proven about a program
- Applications: program verification, program semantics specification
Keyword
A word that is special only in certain contexts
Reserved Word
A special word that cannot be used as a user-defined name
Attributes of Variables
Name, Address, Value, Type, Lifetime, Scope
Binding Times
1. Language Design Time
2. Language Implementation Time
3. Compile Time
4. Load Time
5. Runtime
Static Binding
It first occurs before runtime and remains unchanged throughout program execution
Dynamic Binding
It first occurs during execution or can change during execution of the program.
Categories of Variables by Lifetimes
1. Stack
2. Stack-dynamic
3. Explicit heap-dynamic
4. Implicit heap-dynamic
Stack
Bound to memory cells before execution begins and remains bound to the same memory cell throughout execution
Stack-dynamic
Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
Explicit heap-dynamic
Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution
Implicit heap-dynamic
Allocation and deallocation caused by assignment statements

In formal language theory, a **context-free grammar** (**CFG**) is a certain type of formal grammar: a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements. For example, the rule

replaces with . There can be multiple replacement rules for any given value. For example,

means that can be replaced with either or .

In context-free grammars, all rules are one-to-one, one-to-many, or one-to-none. These rules can be applied regardless of context. The left-hand side of the production rule is also always a nonterminal symbol. This means that the symbol does not

appear in the resulting formal language. So in our case, our language contains the letters and but not .[1]

Rules can also be applied in reverse to check if a string is grammatically correct according to the grammar.

Here is an example context-free grammar that describes all two-letter strings containing the letters and .

If we start with the nonterminal symbol then we can use the rule to turn into . We can then apply

one of the two later rules. For example, if we apply to the first we get . If we then apply to the

second we get . Since both and are terminal symbols, and in context-free grammars terminal symbols never appear on the left hand side of a production rule, there are no more rules that can be applied. This same process can be used, applying the second two rules in different orders in order to get all possible strings within our simple context-free grammar.

Languages generated by context-free grammars are known as context-free languages (CFL). Different context-free grammars can generate the same context-free language. It is important to distinguish properties of the language (intrinsic properties) from properties of a particular grammar (extrinsic properties). The language equality question (do two given context-free grammars generate the same language?) is undecidable.

Context-free grammars arise in linguistics where they are used to describe the structure of sentences and words in natural language, and they were in fact invented by the linguist Noam Chomsky for this purpose, but have not really lived up to their original expectation. By contrast, in computer science, as the use of recursively-defined concepts increased, they were used more and more. In an early application, grammars are used to describe the structure of programming languages. In a newer application, they are used in an essential part of the Extensible Markup Language (XML) called the *Document Type Definition*.[2]

In linguistics, some authors use the term **phrase structure grammar** to refer to context-free grammars, whereby phrase-structure grammars are distinct from dependency grammars. In computer science, a popular notation for context-free grammars is Backus–Naur form, or *BNF*.

Formal definitions[edit]

A context-free grammar G is defined by the 4-tuple:[5]

where

1. V is a finite set; each element is called *a nonterminal character* or a *variable*. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G.
2. Σ is a finite set of *terminals*, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G.
3. R is a finite relation from V to , where the asterisk represents the Kleene star operation. The members of R are called the *(rewrite) rules* or *productions* of the grammar. (also commonly symbolized by a P)
4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

Production rule notation[edit]

A production rule in R is formalized mathematically as a pair , where is a nonterminal and is a string of variables and/or terminals; rather than using ordered pair notation, production rules are usually written using an arrow operator

with α as its left hand side and β as its right hand side: .

It is allowed for β to be the empty string, and in this case it is customary to denote it by ε. The form is called an ε-production.[6]

It is common to list all right-hand sides for the same left-hand side on the same line, using | (the pipe symbol) to separate them.

Rules and can hence be written as . In this case, and is called the first and second alternative, respectively.

Rule application[edit]

For any strings , we say u directly yields v, written as , if with and such that

and . Thus, v is a result of applying the rule to u.

Repetitive rule application[edit]

For any strings we say u **yields** v, written as (or in some textbooks), if such that . In this

case, if (i.e.,), the relation holds. In other words, and are the reflexive transitive

closure (allowing a word to yield itself) and the transitive closure (requiring at least one step) of , respectively.

Context-free language[edit]

The language of a grammar is the set

A language L is said to be a context-free language (CFL), if there exists a CFG G, such that .

Proper CFGs[edit]

A context-free grammar is said to be *proper*,[7] if it has

- no unreachable symbols:
- no unproductive symbols:
- no ε-productions:
- no cycles:

Every context-free grammar can be effectively transformed into a weakly equivalent one without unreachable symbols,[8] a weakly equivalent one without unproductive symbols,[9] and a weakly equivalent one without cycles.[10] Every context-free grammar not producing ε can be effectively transformed into a weakly equivalent one without ε-productions;[11] altogether, every such grammar can be effectively transformed into a weakly equivalent proper CFG.

Example[edit]

The grammar , with productions

$S \to aSa,$
$S \to bSb,$
$S \to \varepsilon,$

is context-free. It is not proper since it includes an ε-production. A typical derivation in this grammar is

$S \to aSa \to aaSaa \to aabSbaa \to aabbaa.$

This makes it clear that . The language is context-free, however it can be proved that it is not regular.

Well-formed parentheses[edit]

The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are

S → SS
S → (S)
S → ()

The first rule allows the S symbol to multiply; the second rule allows the S symbol to become enclosed by matching parentheses; and the third rule terminates the recursion.

Well-formed nested parentheses and square brackets[edit]

A second canonical example is two different kinds of matching nested parentheses, described by the productions:

S → SS
S → ()
S → (S)
S → []
S → [S]

with terminal symbols [ ] ( ) and nonterminal S.

The following sequence can be derived in that grammar:

([[[()() [ ][ ]]]([ ])])

However, there is no context-free grammar for generating all sequences of two different types of parentheses, each separately balanced disregarding the other, but where the two types need not nest inside one another, for example:

[( ])

or

[[[[(((( ]]]]))))(([ ))(([ ))([ ])( ])( ])

A regular grammar[edit]

Every regular grammar is context-free, but not all context-free grammars are regular. The following context-free grammar, however, is also regular.

S → a
S → aS
S → bS

The terminals here are *a* and *b*, while the only nonterminal is S. The language described is all nonempty strings

of s and s that end in .

This grammar is regular: no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at the same end of the right-hand side.

Every regular grammar corresponds directly to a nondeterministic finite automaton, so we know that this is a regular language.

Using pipe symbols, the grammar above can be described more tersely as follows:

S → a | aS | bS

Matching pairs[edit]

In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:

S → aSb
S → ab

This grammar generates the language , which is not regular (according to the pumping lemma for regular languages).

The special character ε stands for the empty string. By changing the above grammar to

S → aSb | ε

we obtain a grammar generating the language instead. This differs only in that it contains the empty string while the original grammar did not.

Algebraic expressions[edit]

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. S → x
2. S → y
3. S → z
4. S → S + S
5. S → S - S
6. S → S * S
7. S → S / S
8. S → (S)

This grammar can, for example, generate the string

(x + y) * x - z * y / (x + x)

as follows:

S (the start symbol)
→ S - S (by rule 5)
→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)
→ (S) * S - S / S (by rule 8, applied to the leftmost S)
→ (S) * S - S / (S) (by rule 8, applied to the rightmost S)
→ (S + S) * S - S / (S) (etc.)
→ (S + S) * S - S * S / (S)
→ (S + S) * S - S * S / (S + S)
→ (x + S) * S - S * S / (S + S)
→ (x + y) * S - S * S / (S + S)
→ (x + y) * x - S * y / (S + S)
→ (x + y) * x - S * y / (x + S)
→ (x + y) * x - z * y / (x + S)
→ (x + y) * x - z * y / (x + x)

Note that many choices were made underway as to which rewrite was going to be performed next. These choices look quite arbitrary. As a matter of fact, they are, in the sense that the string finally generated is always the same. For example, the second and third rewrites

→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)

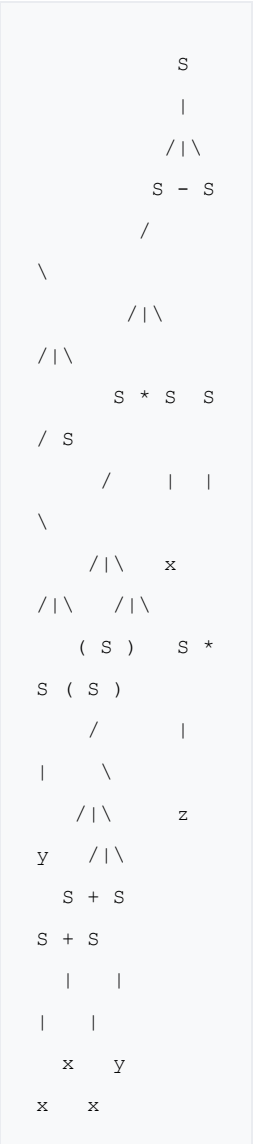
could be done in the opposite order:

→ S - S / S (by rule 7, applied to the rightmost S)

→ S * S - S / S (by rule 6, applied to the leftmost S)

Also, many choices were made on which rule to apply to each selected S. Changing the choices made and not only the order they were made in usually affects which terminal string comes out at the end.

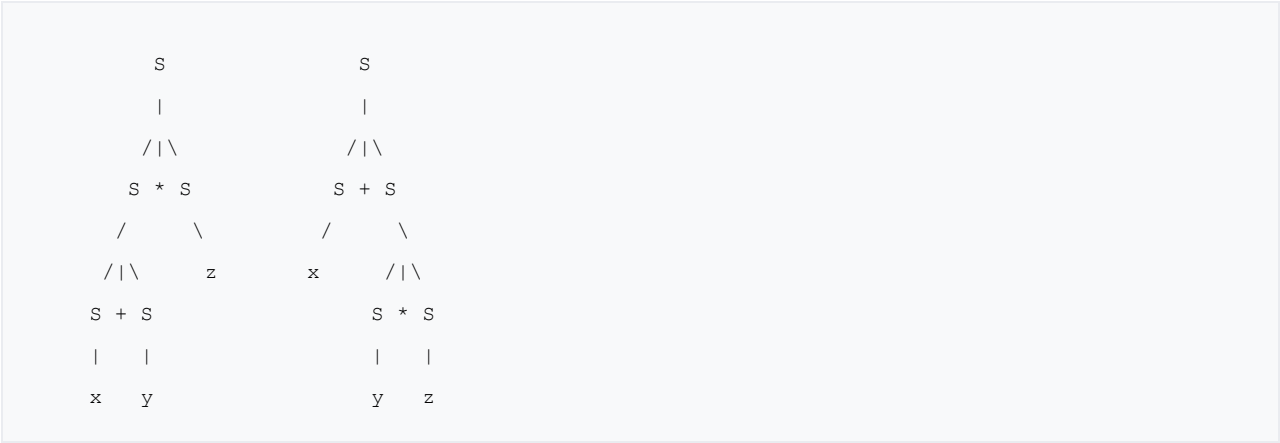
Let's look at this in more detail. Consider the parse tree of this derivation:



Starting at the top, step by step, an S in the tree is expanded, until no more unexpanded Ses (nonterminals) remain. Picking a different order of expansion will produce a different derivation, but the same parse tree. The parse tree will only change if we pick a different rule to apply at some position in the tree.

But can a different parse tree still produce the same terminal string, which is `( x + y ) * x − z * y / ( x + x )` in this case? Yes, for this particular grammar, this is possible. Grammars with this property are called ambiguous.

For example, `x + y * z` can be produced with these two different parse trees:



However, the *language* described by this grammar is not inherently ambiguous: an alternative, unambiguous grammar can be given for the language, for example:

T → x
T → y
T → z
S → S + T
S → S - T
S → S * T
S → S / T
T → (S)
S → T

(once again picking S as the start symbol). This alternative grammar will produce `x + y * z` with a parse tree similar to the left one above, i.e. implicitly assuming the association `(x + y) * z`, which is not according to standard operator precedence. More elaborate, unambiguous and context-free grammars can be constructed that produce parse trees that obey all desired operator precedence and associativity rules.

Further examples[edit]

Example 1[edit]

A context-free grammar for the language consisting of all strings over {a,b} containing an unequal number of a's and b's:

S → U | V
U → TaU | TaT | UaT
V → TbV | TbT | VbT
T → aTbT | bTaT | ε

Here, the nonterminal T can generate all strings with the same number of a's as b's, the nonterminal U generates all strings with more a's than b's and the nonterminal V generates all strings with fewer a's than b's. Omitting the third alternative in the rule for U and V doesn't restrict the grammar's language.

Example 2[edit]

Another example of a non-regular language is . It is context-free as it can be generated by the following context-free grammar:

S → bSbb | A
A → aA | ε

Context – free grammar

In formal language theory, a **context-free grammar** (**CFG**) is a certain type of formal grammar: a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements. For example, the rule

$A \to \alpha$

replaces A with α. There can be multiple replacement rules for any given value. For example,

$A \to \alpha$

$A \to \beta$

means that A can be replaced with either α or β.

In context-free grammars, all rules are one-to-one, one-to-many, or one-to-none. These rules can be applied regardless of context. The left-hand side of the production rule is also always a nonterminal symbol. This means that the symbol does not appear in the resulting formal language. So in our case, our language contains the letters α and β but not A.[1]

Rules can also be applied in reverse to check if a string is grammatically correct according to the grammar.

Here is an example context-free grammar that describes all two-letter strings containing the letters α and β.

$S \to AA$

$A \to \alpha$

$A \to \beta$

If we start with the nonterminal symbol S then we can use the rule $S \to AA$ to turn S into AA. We can then apply one of the two later rules. For example, if we apply $A \to \beta$ to the first A we get βA. If we then apply $A \to \alpha$ to the second A we get $\beta\alpha$. Since both α and β are terminal symbols, and in context-free grammars terminal symbols never appear on the left hand side of a production rule, there are no more rules that can be applied. This same process can be used, applying the second two rules in different orders in order to get all possible strings within our simple context-free grammar.

Languages generated by context-free grammars are known as context-free languages (CFL). Different context-free grammars can generate the same context-free language. It is important to distinguish properties of the language (intrinsic properties) from properties of a particular grammar (extrinsic properties). The language equality question (do two given context-free grammars generate the same language?) is undecidable.

Context-free grammars arise in linguistics where they are used to describe the structure of sentences and words in natural language, and they were in fact invented by the linguist Noam Chomsky for this purpose, but have not really lived up to their original expectation. By contrast, in computer science, as the use of recursively-defined concepts increased, they were used more and more. In an early application, grammars are used to describe the structure of programming languages. In a newer application, they are used in an essential part of the Extensible Markup Language (XML) called the *Document Type Definition*.[2]

In linguistics, some authors use the term **phrase structure grammar** to refer to context-free grammars, whereby phrase-structure grammars are distinct from dependency grammars. In computer science, a popular notation for context-free grammars is Backus–Naur form, or *BNF*.

Formal definitions [edit]

A context-free grammar G is defined by the 4-tuple:[5]

$G = (V, \Sigma, R, S)$ where

1. V is a finite set; each element $v \in V$ is called *a nonterminal character* or a *variable*. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by G.
2. Σ is a finite set of *terminal*s, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G.
3. R is a finite relation from V to $(V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation. The members of R are called the *(rewrite) rule*s or *production*s of the grammar. (also commonly symbolized by a P)
4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V.

Production rule notation [edit]

A production rule in R is formalized mathematically as a pair $(\alpha, \beta) \in R$, where $\alpha \in V$ is a nonterminal and $\beta \in (V \cup \Sigma)^*$ is a string of variables and/or terminals; rather than using ordered pair notation, production rules are usually written using an arrow operator with α as its left hand side and β as its right hand side: $\alpha \to \beta$.

It is allowed for β to be the empty string, and in this case it is customary to denote it by ε. The form $\alpha \to \varepsilon$ is called an ε-production.[6]

It is common to list all right-hand sides for the same left-hand side on the same line, using | (the pipe symbol) to separate them. Rules $\alpha \to \beta_1$ and $\alpha \to \beta_2$ can hence be written as $\alpha \to \beta_1 \mid \beta_2$. In this case, β_1 and β_2 is called the first and second alternative, respectively.

Rule application [edit]

For any strings $u, v \in (V \cup \Sigma)^*$, we say u directly yields v, written as $u \Rightarrow v$, if $\exists (\alpha, \beta) \in R$ with $\alpha \in V$ and $u_1, u_2 \in (V \cup \Sigma)^*$ such that $u = u_1 \alpha u_2$ and $v = u_1 \beta u_2$. Thus, v is a result of applying the rule (α, β) to u.

Repetitive rule application [edit]

For any strings $u, v \in (V \cup \Sigma)^*$, we say u **yields** v, written as $u \overset{*}{\Rightarrow} v$ (or $u \Rightarrow\Rightarrow v$ in some textbooks), if $\exists k \geq 1 \, \exists u_1, \cdots, u_k \in (V \cup \Sigma)^*$ such that $u = u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k = v$. In this case, if $k \geq 2$ (i.e., $u \neq v$), the relation $u \overset{+}{\Rightarrow} v$ holds. In other words, $\left(\overset{*}{\Rightarrow}\right)$ and $\left(\overset{+}{\Rightarrow}\right)$ are the reflexive transitive closure (allowing a word to yield itself) and the transitive closure (requiring at least one step) of $(\Rightarrow)$, respectively.

Context-free language [edit]

The language of a grammar $G = (V, \Sigma, R, S)$ is the set

$$L(G) = \{w \in \Sigma^* : S \overset{*}{\Rightarrow} w\}$$

A language L is said to be a context-free language (CFL), if there exists a CFG G, such that $L = L(G)$.

Proper CFGs [edit]

A context-free grammar is said to be *proper*,[7] if it has

- no unreachable symbols: $\forall N \in V : \exists \alpha, \beta \in (V \cup \Sigma)^* : S \overset{*}{\Rightarrow} \alpha N \beta$
- no unproductive symbols: $\forall N \in V : \exists w \in \Sigma^* : N \overset{*}{\Rightarrow} w$
- no ε-productions: $\neg \exists N \in V : (N, \varepsilon) \in R$
- no cycles: $\neg \exists N \in V : N \overset{+}{\Rightarrow} N$

Every context-free grammar can be effectively transformed into a weakly equivalent one without unreachable symbols,[8] a weakly equivalent one without unproductive symbols,[9] and a weakly equivalent one without cycles.[10] Every context-free grammar not producing ε can be effectively transformed into a weakly equivalent one without ε-productions;[11] altogether, every such grammar can be effectively transformed into a weakly equivalent proper CFG.

Example [edit]

The grammar $G = (\{S\}, \{a, b\}, P, S)$, with productions

$S \to aSa$,
$S \to bSb$,
$S \to \varepsilon$,

is context-free. It is not proper since it includes an ε-production. A typical derivation in this grammar is

$S \to aSa \to aaSaa \to aabSbaa \to aabbaa$.

This makes it clear that $L(G) = \{ww^R : w \in \{a, b\}^*\}$. The language is context-free, however it can be proved that it is not regular.

Well-formed parentheses [edit]

The canonical example of a context free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol S. The production rules are

S → SS
S → (S)
S → ()

The first rule allows the S symbol to multiply; the second rule allows the S symbol to become enclosed by matching parentheses; and the third rule terminates the recursion.

Well-formed nested parentheses and square brackets [edit]

A second canonical example is two different kinds of matching nested parentheses, described by the productions:

S → SS
S → ()
S → (S)
S → []
S → [S]

with terminal symbols [ ] ( ) and nonterminal S.

The following sequence can be derived in that grammar:

([[[()() [ ][ ]]]([ ])])

However, there is no context-free grammar for generating all sequences of two different types of parentheses, each separately balanced disregarding the other, but where the two types need not nest inside one another, for example:

[( ])

A regular grammar [edit]

Every regular grammar is context-free, but not all context-free grammars are regular. The following context-free grammar, however, is also regular.

S → a
S → aS
S → bS

The terminals here are *a* and *b*, while the only nonterminal is S. The language described is all nonempty strings of as and bs that end in a.

This grammar is regular: no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at the same end of the right-hand side.

Every regular grammar corresponds directly to a nondeterministic finite automaton, so we know that this is a regular language.

Using pipe symbols, the grammar above can be described more tersely as follows:

S → a | aS | bS

Matching pairs [edit]

In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:

S → aSb
S → ab

This grammar generates the language $\{a^n b^n : n \geq 1\}$, which is not regular (according to the pumping lemma for regular languages).

The special character ε stands for the empty string. By changing the above grammar to

S → aSb | ε

we obtain a grammar generating the language $\{a^n b^n : n \geq 0\}$ instead. This differs only in that it contains the empty string while the original grammar did not.

Algebraic expressions [edit]

Here is a context-free grammar for syntactically correct infix algebraic expressions in the variables x, y and z:

1. S → x
2. S → y
3. S → z
4. S → S + S
5. S → S - S
6. S → S * S
7. S → S / S
8. S → (S)

This grammar can, for example, generate the string

(x + y) * x - z * y / (x + x)

as follows:

S (the start symbol)
→ S - S (by rule 5)
→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)
→ (S) * S - S / S (by rule 8, applied to the leftmost S)
→ (S) * S - S / (S) (by rule 8, applied to the rightmost S)
→ (S + S) * S - S / (S) (etc.)
→ (S + S) * S - S * S / (S)
→ (S + S) * S - S * S / (S + S)
→ (x + S) * S - S * S / (S + S)
→ (x + y) * S - S * S / (S + S)
→ (x + y) * x - S * y / (S + S)
→ (x + y) * x - S * y / (x + S)
→ (x + y) * x - z * y / (x + S)
→ (x + y) * x - z * y / (x + x)

→ (x + y) * x - z * y / (x + x)

Note that many choices were made underway as to which rewrite was going to be performed next. These choices look quite arbitrary. As a matter of fact, they are, in the sense that the string finally generated is always the same. For example, the second and third rewrites

→ S * S - S (by rule 6, applied to the leftmost S)
→ S * S - S / S (by rule 7, applied to the rightmost S)

could be done in the opposite order:

→ S - S / S (by rule 7, applied to the rightmost S)
→ S * S - S / S (by rule 6, applied to the leftmost S)

Also, many choices were made on which rule to apply to each selected `S`. Changing the choices made and not only the order they were made in usually affects which terminal string comes out at the end.

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