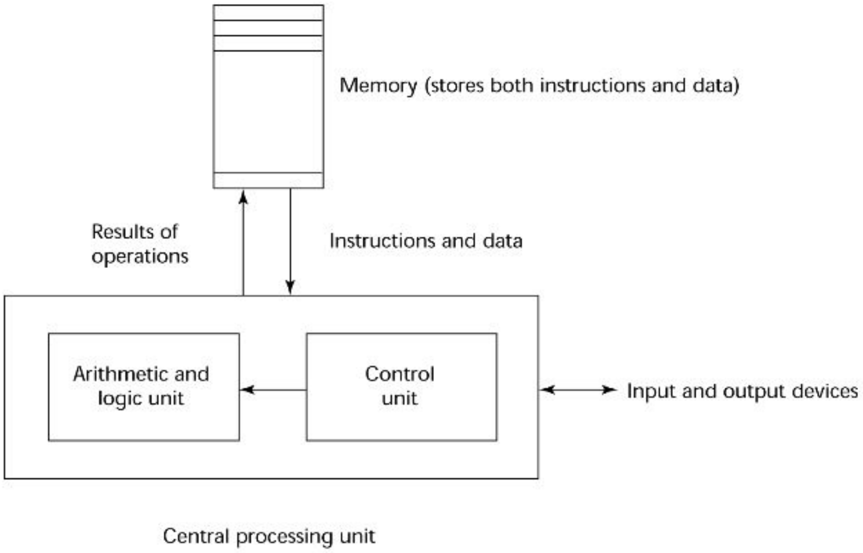
- We discussed reasons for studying programming languages, such as expanded capacity for expression, accelerated learning curve, etc.
- Domains of programming languages (scientific, business, AI, systems, scripting)
    - **Scientific applications**
        - Large numbers of floating point computations; use of arrays
        - Fortran
    - **Business applications**
        - Produce reports, use decimal numbers and characters
        - COBOL
    - **Artificial intelligence**
        - Symbols rather than numbers manipulated; use of linked lists
        - LISP
    - **Systems programming**
        - Need efficiency because of continuous use
        - C
    - **Web Software**
        - Eclectic collection of languages: markup (e.g., HTML), scripting (e.g., PHP), general-purpose (e.g., Java)
- How to evaluate languages (issues to consider: control constructs, data types (and definability), syntax, expressivity)
    - **Readability**
        - The ease with which programs can be read and understood
        - **Simplicity**
            - A manageable set of features and constructs
            - Minimal feature multiplicity
            - Minimal operator overloading
        - **Orthogonality**
            - A relatively small set of primitive constructs can be combined in a relatively small number of ways
            - Every possible combination is legal
        - **Data Types:** Adequate, predefined data types
        - **Syntax Considerations**
            - Identifier forms: flexible composition
            - Special words & methods of forming compound statements
            - Form & meaning: self descriptive constructs meaningful words
    - **Writability**
        - The ease with which a language can be used to create programs
        - **Simplicity and orthogonality:** Few constructs, small number of primitives, small set of rules for combining them
        - **Support for abstraction:** Ability to define & use complex structures or operations in ways that allow details to be ignored
        - **Expressivity**
            - Set of relatively convenient ways of specifying operations
            - Strength and numbers of operators and predefined functions
    - **Reliability**
        - Conformance to specifications (i.e., performs to its specifications)

    - **Type checking:** testing for type errors
    - **Exception handling**: intercept run-time errors & take corrective measures
    - **Aliasing:** presence of two or more distinct referencing methods for the same memory location
    - **^Readability and Writability^**: a language that doesn't support "natural" ways of expressing an algorithm will require the use of "unnatural" approaches and reduce reliability
  - **Cost (different kinds): the ultimate total cost**
    - Training programmers to use the language
    - Writing programs (closeness to particular
    - applications)
    - Compiling programs
    - Executing programs
    - Language implementation system: availability of free compilers
    - **Reliability:** poor reliability leads to high costs
    - Maintaining programs
  - Other criteria include :
    - Portability: ease of running on different platforms moved from one implementation to another
    - Generality: applicability to a wide range of applications
    - Well-definedness: completeness of language and original definition
- We looked at design influences on languages (architectures, methodology...)
  - Computer Architecture
    - Languages are developed around the prevalent computer architecture, known as the von Neumann architecture
      - **Von Neumann Architecture**
        - Data and programs stored in memory
        - Memory is separate from CPU : Instructions and data are piped from memory to CPU
        - Basis for imperative languages
          - Variables model memory cells
          - Assignment statements model piping
          - Iteration is efficient



- Program Design Methodologies
    - New software development methodologies (e.g. object-oriented software development) led to new programming paradigms and by extension, new programming languages
    - 1950s and early 1960s: Simple applications; worry about machine efficiency
    - Late 1960s: People efficiency became important; readability, better control structures
        - structured programming
        - top-down design and step-wise refinement
    - Late 1970s: Process-oriented to data-oriented
        - data abstraction
    - Middle 1980s: Object-oriented programming
        - Data abstraction + inheritance + polymorphism

- Language paradigms (imperative, functional, declarative, object-oriented,...)
    - **Imperative**:
        - Central features are variables, assignment statements, and iteration
        - Include scripting languages, visual languages, oop languages
        - C, Java, Perl, JavaScript, Visual BASIC .NET, C++
        - Running on von Neumann architecture requires memory divides data between static, run-time stack, and heap.
        - OOP is included in C++, Java, and other imperative languages.
    - **Functional**:
        - LISP (AI), Scheme, Racket, ML.
        - Functions and parameter passing are primary form of data transformation. Original data is not usually modified.
        - run-time stack is primary memory region
        - Any program that can be written in an imperative style can also be written in functional style.
    - **Logical**:

        - Prolog
        - Database of rules with “programs” that obtain true/false results from questions queried from database
        - Rules specified in no particular order.
    - **Declarative**:
        - Declarative programming is a non-imperative style of programming in which programs describe their desired results without explicitly listing commands or steps that must be performed. Functional and logical programming languages are characterized by a declarative programming style.
    - **Object-Oriented**:
        - A subset of imperative languages
        - Currently incorporated in many languages in other models
    - Markup/programming hybrid
        - JSTL, XSTL, HTML
- Implementation issues (interface to OS, process of compilation/interpretation, environments)
    - **Compiled**—fast execution as runs in native machine language
        - Compiled prior to execution
    - **Interpreted**—slow execution as each statement evaluated at run-time
        - Compiles during runtime
    - **Hybrid**—compiled to intermediate code (example, Java bytecode), then intermediate code evaluated at runtime.
- Language origins, history and impact (look back over the details and concentrate on the ones the instructor highlighted)
    - What was unique about **Plankalkul**?
        - First programming language
        - Unknown until early 1970s
        - Never implemented
        - Advanced data structures
            - floating point, arrays, records (similar to structs in C)
        - Invariants
    - **FORTRAN**
        - Scientific applications
            - Large numbers of floating point computations; use of arrays
        - History of Fortran (from Sebesta slides)
        - Fortran I: First implemented version of Fortran
            - Names could have up to six characters
            - Post-test counting loop (DO)
            - Formatted I/O
            - User-defined subprograms
            - Three-way selection statement (arithmetic IF)
            - No data typing statements
            - No separate compilation
            - Compiler released in April 1957, after 18 worker-years of effort
            - Programs larger than 400 lines rarely compiled correctly, mainly due to poor reliability of IBM 704
            - Code was very fast
            - Quickly became widely used
        - Fortran II

    - Distributed in 1958
      - Independent compilation
      - Fixed the bugs
  - Fortran IV
    - Evolved during 1960-62
      - Explicit type declarations
      - Logical selection statement
      - Subprogram names could be parameters
      - ANSI standard in 1966
  - Fortran 77
    - Became the new standard in 1978
      - Character string handling
      - Logical loop control statement
      - IF-THEN-ELSE statement
  - Fortran 90
    - Most significant changes from Fortran 77
      - Modules
      - Dynamic arrays
      - Pointers
      - Recursion
      - CASE statement
      - Parameter type checking
  - Fortran 95 – relatively minor additions, plus some deletions
  - Fortran 2003 – support for OOP, procedure pointers, interoperability with C
  - Fortran 2008 – blocks for local scopes, co-arrays, Do Concurrent
- **Lisp**
  - Artificial intelligence
    - Symbols rather than numbers manipulated; use of linked lists
  - Only two data types: atoms and lists
  - Syntax is based on lambda calculus
  - Pioneered functional programming
    - No need for variables or assignment
    - Control via recursion and conditional expressions
- **Algol 60**
  - First language to introduce structure programming concepts
  - Prior language FORTRAN used ‘goto’ statements
  - Most modern languages are considered descendants of Algol
  - The result of efforts to design a universal language
  - New concepts:
    - Compound statements
    - Unlimited-dimension arrays
    - Arbitrary length identifiers
    - Block structure
    - Recursive procedure
    - Stack-dynamic arrays
    - BNF notation
- **COBOL**
  - Business applications

    - Produce reports, use decimal numbers and characters
  - Common Business-Oriented Language
  - Still in dominates business computing
  - Based on FLOW-MATIC
  - FLOW-MATIC features
    - Names up to 12 characters, with embedded hyphens
    - English names for arithmetic operators (no arithmetic expressions)
    - Data and code were completely separate
    - The first word in every statement was a verb
- **BASIC**
  - Beginner's All-purpose Symbolic Instruction Code
  - For "terminals" connected to a mainframe computer, which were teletypewriters (think latest breaking news wire services). Print (portion of) program on terminal, make change, reprint to see changes.
  - Could store program by having it punch to paper tape, 1-inch wide, six-holded, tear-off "printer"
- **PL/I**
  - First unit-level concurrency
  - First exception handling
  - Switch-selectable recursion
  - First pointer data type
  - First array cross sections
  - Concerns
    - Many new features were poorly designed
    - Too large and too complex
- **Simula 67 and Algol 68** (what was conspicuous about them/what did they contribute?)
  - Simula 67
    - Primary Contributions
      - Coroutines **-** a kind of subprogram
      - Classes, objects, and inheritance
  - Algol 68
    - Contributions
      - User-defined data structures
      - Reference types
      - Dynamic arrays (called flex arrays)
    - Design is based on the concept of orthogonality
      - A few basic concepts, plus a few combining mechanisms
    - Had strong influence on subsequent languages, especially Pascal, C, and Ada
- **Pascal**
  - Designed for teaching structured programming
  - Small, simple, nothing really new
- **C**
  - Designed for systems programming
    - Need efficiency because of continuous use
  - Powerful set of operators, but poor type checking
  - Initially spread through UNIX
- **Perl**
  - Variables are statically typed but implicitly declared

        - Three distinctive namespaces, denoted by the first character of a variable's name
        - Powerful, but somewhat dangerous
        - Gained widespread use for CGI programming on the Web
        - Also used for a replacement for UNIX system administration language
    - **Prolog**
        - Based on formal logic
        - Non-procedural
        - Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries
        - Comparatively inefficient
        - Few application areas
    - **Ada**
        - Contributions
            - Packages - support for data abstraction
            - Exception handling - elaborate
            - Generic program units
            - Concurrency - through the tasking model
        - First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed
    - **Smalltalk**
        - First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic binding)
        - Pioneered the graphical user interface design
        - Promoted OOP
        - Everything is an object
    - **C++**
        - Evolved from C and SIMULA 67
        - Facilities for object-oriented programming, taken partially
        - from SIMULA 67
        - A large and complex language, in part because it supports both procedural and OO programming
        - Rapidly grew in popularity, along with OOP
        - ANSI standard approved in November 1997
        - Microsoft's version: MC++
            - Properties, delegates, interfaces, no multiple inheritance
    - **Java**
        - C and C++ were not satisfactory for embedded electronic devices
        - Based on C++
            - Significantly simplified (does not include struct, union, enum, pointer arithmetic, and half of the assignment coercions of C++)
            - Supports only OOP
            - Has references, but not pointers
            - Includes support for applets and a form of concurrence
    - What are several of the scripting languages?
        - Perl, Javascript, PHP, Python, Ruby, Lua (information about each can be found in the Chapter 2 Sebesta slides 60-63. Sebesta slides are found on Blackboard > Course Documents > Lecture slides, videos, and documents > Sebesta Slides 11ed)
    - **C#**

        - Based on C++ , Java, and Delphi
        - Includes pointers, delegates, properties, enumeration types, a limited kind of dynamic typing, and anonymous types
        - Is evolving rapidly
- Languages for describing languages: **BNF, EBNF** (know how to use EBNF to describe a basic control construct in a typical language)
    - Adds other symbols to simplify expressions:
    - item? or [item] means the item is optional
    - item* or {item} means zero or more occurrences of item are allowed
    - item+ means one or more occurrences of an item are allowed
- Syntax vs. semantics
    - **Syntax**
        - The syntax of a programming language determines the well-formed or grammatically correct programs of the language. It is specified by a grammar.
        - How programs look.
    - **Semantics**
        - The act of deriving meaning from what is written is semantics. It refers to what the program will do when it is executed.
        - Describes how or whether syntactically correct programs will execute.
        - How programs work.
- Formal terms: alphabet, sentence, language, token, lexeme, **grammar, derivations, parse trees**
    - **Alphabet**
        - A set of all the symbols that act as letters
    - **Sentence**
        - A string of characters over some alphabet
    - **Language**
        - A set of sentences
    - **Lexeme**
        - Lowest level syntactic unit of a language
    - **Token**
        - Category of lexemes
    - **Grammar**
    - **Derivations**
        - A repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
    - **Parse tree**
        - A hierarchical representation of a derivation
- Four classes of language: in particular the ones we use, context-free (almost [all the time]) **She was unsure about this, said this is what we should assume this question meant**
    - Imperative
    - Functional
    - Logical
    - OOP
- Ambiguity in grammars, an example
    - Ambiguity
        - If and only if it generates a sentential form that has two or more distinct parse trees.
        - With a glance, you can identify ambiguity from if it is recursive. Recursivity often means ambiguous, since more than one parse tree can be made.

- Recursion in grammars, an example
- Producing parse trees
- Describe kinds of algorithms for parsing: top-down (recursive-descent), bottom-up (shift-reduce)
    - **Recursive descent parsing**
        - There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
        - EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals
        - The coding process when there is only one RHS:
        - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
        - For each nonterminal symbol in the RHS, call its associated parsing subprogram
        - This particular routine does not detect errors
        - Convention: Every parsing routine leaves the next token in nextToken
    - **Shift-Reduce Algorithms**
        - –Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
        - –Shift is the action of moving the next token to the top of the parse stack
- **Operational semantics**
    - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- **Denotational semantics**
    - Based on recursive function theory
    - The most abstract semantics description method
    - The process of building a denotational specification for a language:
        - Define a mathematical object for each language entity
        - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- Transition diagrams used in building a lexical analyzer
- Variables=(name,address,value) in C
- Basic information about these data types in C:
    - Integer
        - 4 bytes, declared “int”
    - Floating-point
        - 4 bytes
    - Double
        - 8 bytes
    - Boolean
        - 1 byte
    - Character
        - 1 byte, declared “char”
    - Character strings: array or built-in or object, null-termination and associated errors, regular expressions for pattern matching
    - Character strings: static or dynamic length and associated bookkeeping, three styles of storage management for dynamic
- **C language**

- compiling and running a program
  - Most basic compilation is with: gcc fileName.c
    - Executable file will be called a.out
    - Can execute by calling a
  - For p1 we used: gcc -lm -ansi -Wall -o p1 p1.c
    - gcc: invoke GNU C compiler
    - -ansi: use ANSI-C version (an older version, that, among other things, does not permit comments of type "//")
    - -lm: link in the math library
    - -Wall: show all warnings
    - -o p1: name the executable file "p1" (this is the output of a successful compilation--if not explicitly named, the executable is "a.out")
    - P1.c: the C source file being compiled
- getting arguments from command line
  - Arguments on the command line: p1a 1000 0.18 100
    - argv[0] = p1a, argv[1] = 1000 etc.
- printf and scanf I/O
- variables: types, address of variables vs. value stored in variable, initialization
- pointers: declaring, types, dereferencing/indirection
  - '*' symbol is used to point to a variable, it is also the symbol to dereference a variable.
- struct, typedef: what are they used for, how to declare and use
- passing by value vs. passing by reference to functions
  - Passing by values passes the value of a variable to another..
  - Passing by reference uses '&' and passes the memory location.
- file I/O
  - We did not go over this
- write a short C program in 15 minutes (the likes of our p1.c)

Left and Right associativity:

Eg. Given the statement 3 - 2 -1

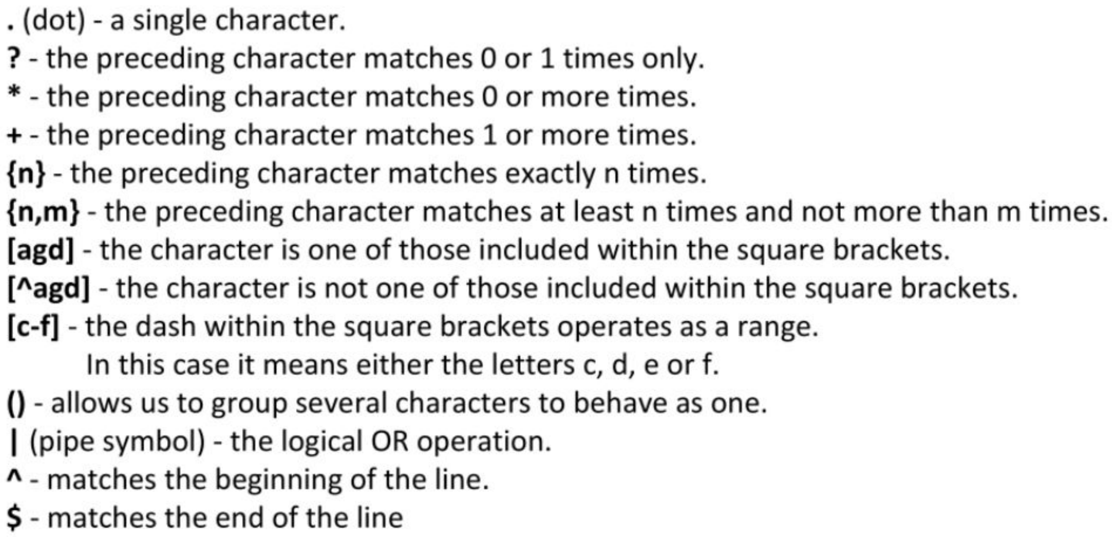
Left associativity: (3 - 2) - 1

Right associativity: 3 - (2 - 1)

Regex101.com to check regex solutions

Prefix / infix converter: https://www.web4college.com/converters/infix-to-postfix-prefix.php

Regular Expressions:



. (dot) - a single character.

? - the preceding character matches 0 or 1 times only.

* - the preceding character matches 0 or more times.

+ - the preceding character matches 1 or more times.

{n} - the preceding character matches exactly n times.

{n,m} - the preceding character matches at least n times and not more than m times.

[agd] - the character is one of those included within the square brackets.

[^agd] - the character is not one of those included within the square brackets.

[c-f] - the dash within the square brackets operates as a range.
In this case it means either the letters c, d, e or f.

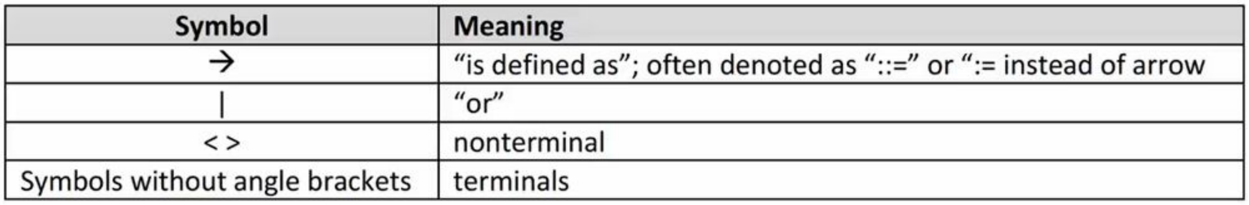
() - allows us to group several characters to behave as one.

| (pipe symbol) - the logical OR operation.

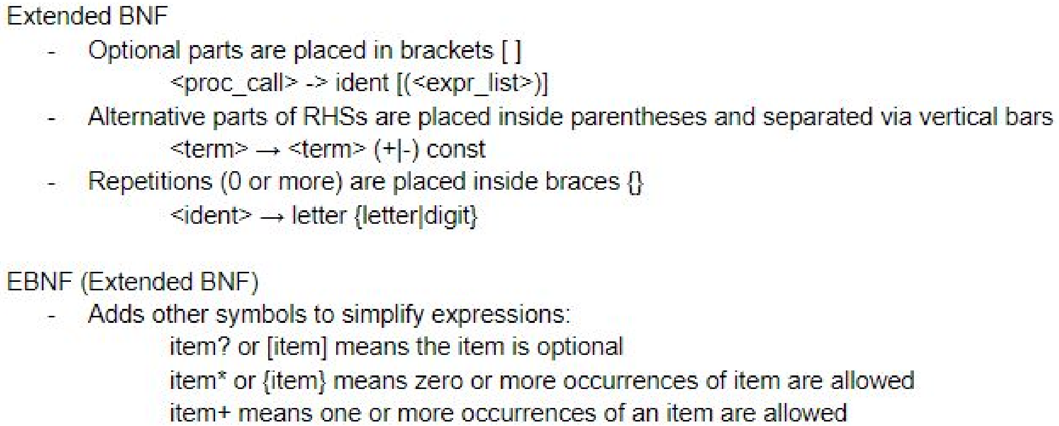
^ - matches the beginning of the line.

$ - matches the end of the line

BNF and EBNF References:



Symbol	Meaning
→	“is defined as”; often denoted as “::=” or “:= instead of arrow
|	“or”
< >	nonterminal
Symbols without angle brackets	terminals

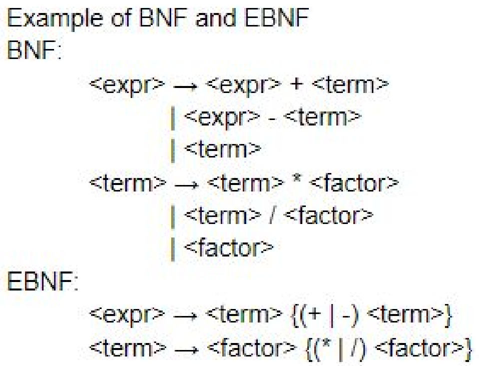


Extended BNF

- Optional parts are placed in brackets [ ]
  - <proc_call> -> ident [(<expr_list>)]
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  - <term> → <term> (+|-) const
- Repetitions (0 or more) are placed inside braces {}
  - <ident> → letter {letter|digit}

EBNF (Extended BNF)

- Adds other symbols to simplify expressions:
  - item? or [item] means the item is optional
  - item* or {item} means zero or more occurrences of item are allowed
  - item+ means one or more occurrences of an item are allowed



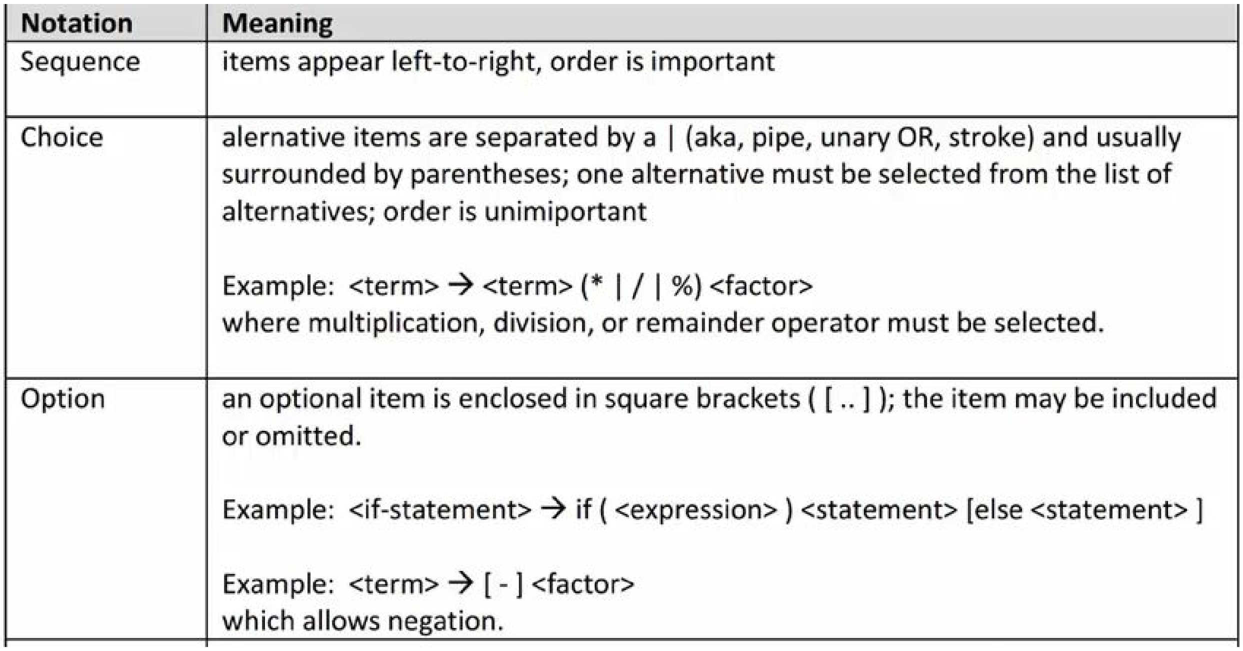
Example of BNF and EBNF

BNF:

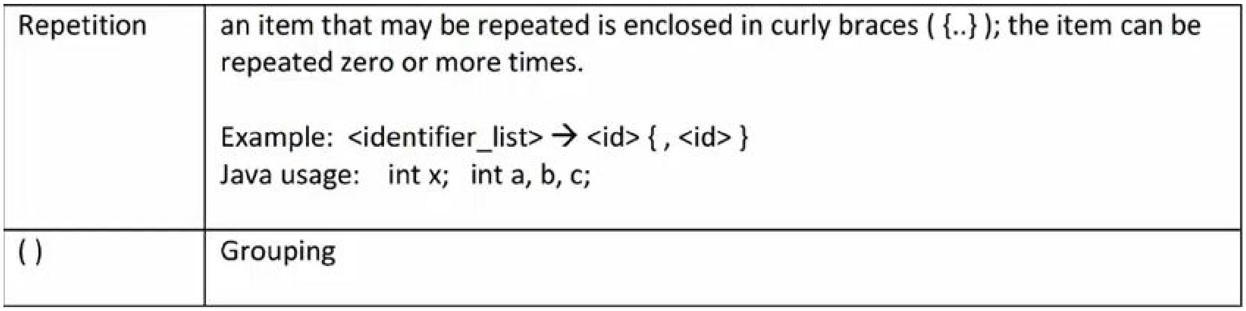
```
    <expr> → <expr> + <term>
             | <expr> - <term>
             | <term>
    <term> → <term> * <factor>
             | <term> / <factor>
             | <factor>
```

EBNF:

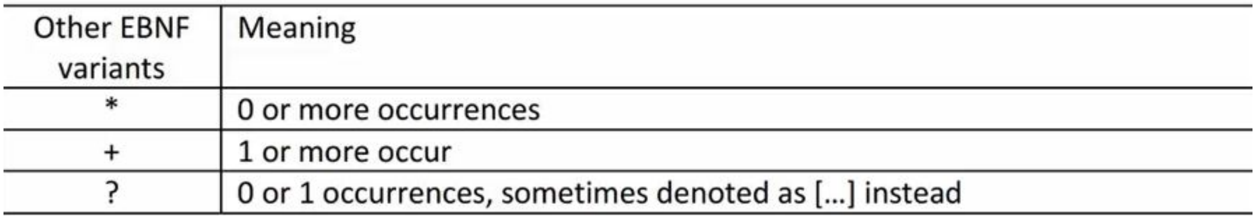
```
    <expr> → <term> {(+ | -) <term>}
    <term> → <factor> {(* | /) <factor>}
```



Notation	Meaning
Sequence	items appear left-to-right, order is important
Choice	alernative items are separated by a | (aka, pipe, unary OR, stroke) and usually surrounded by parentheses; one alternative must be selected from the list of alternatives; order is unimiportant Example: <term> → <term> (* | / | %) <factor> where multiplication, division, or remainder operator must be selected.
Option	an optional item is enclosed in square brackets ([..]); the item may be included or omitted. Example: <if-statement> → if (<expression>) <statement> [else <statement>] Example: <term> → [-] <factor> which allows negation.



Repetition	an item that may be repeated is enclosed in curly braces ({..}); the item can be repeated zero or more times. Example: <identifier_list> → <id> { , <id> } Java usage: int x; int a, b, c;
( )	Grouping



Other EBNF variants	Meaning
*	0 or more occurrences
+	1 or more occur
?	0 or 1 occurrences, sometimes denoted as […] instead