# Chapter 3

## (a) Syntax



## Some Preliminaries

- For the next several weeks we'll look at how one can define a programming language.
- What is a language, anyway?

  Language is a system of gestures, grammar, signs, sounds, symbols, or words, which is used to represent and communicate concepts, ideas, meanings, and thoughts.
- Human language is a way to communicate representations from one (human) mind to another
- What about a programming language?

  A way to communicate representations (e.g., of data or a procedure) between human minds and/or machines.



## Introduction

We usually break down the problem of defining a programming language into two parts.

- defining the PL's syntax
- defining the PL's semantics

*Syntax* - the **form** or structure of the expressions, statements, and program units

*Semantics* - the **meaning** of the expressions, statements, and program units.

Note: There is not always a clear boundary between the two.

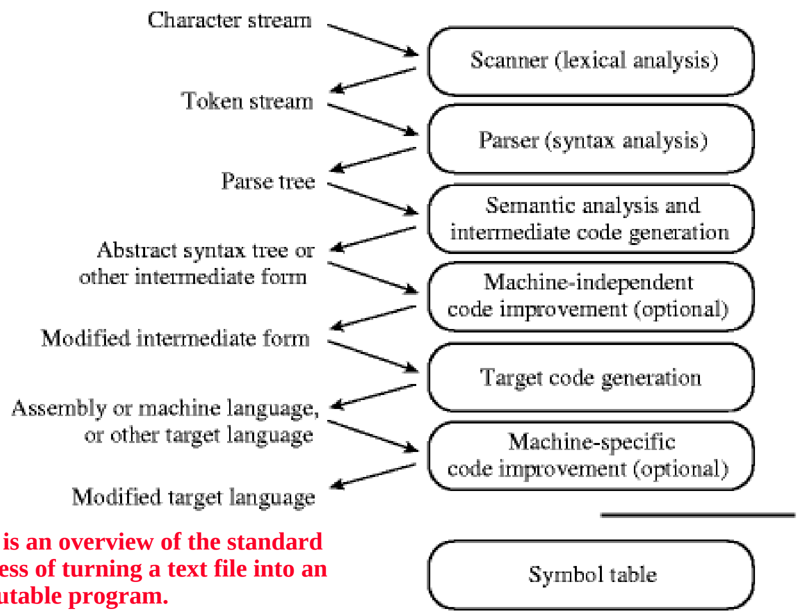


## Why and How

**Why?** We want specifications for several communities:

- Other language designers
- Implementers
- Machines?
- Programmers (the users of the language)

**How?** One ways is via natural language descriptions (e.g., user's manuals, text books) but there are a number of techniques for specifying the syntax and semantics that are more formal.




**This is an overview of the standard process of turning a text file into an executable program.**



# Syntax Overview

- **Language preliminaries**
- **Context-free grammars and BNF**
- **Syntax diagrams**



# Introduction

A *sentence* is a string of characters over some alphabet.

A *language* is a set of sentences.

A *lexeme* is the lowest level syntactic unit of a language (e.g., *, *sum*, *begin*).

A *token* is a category of lexemes (e.g., *identifier*).

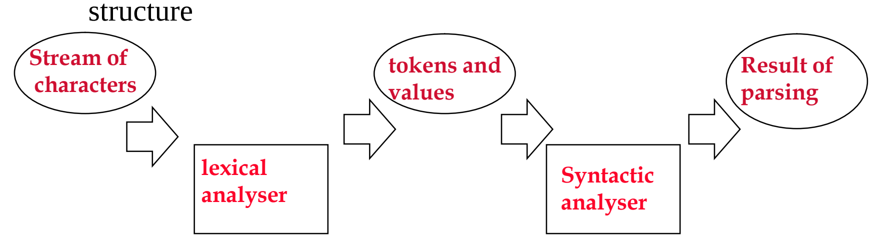
Formal approaches to describing syntax:

1. Recognizers - used in compilers
2. Generators - what we'll study



# Lexical Structure of Programming Languages

- The structure of its lexemes (words or tokens)
  - token is a category of lexeme
- The scanning phase (lexical analyser) collects characters into tokens
- Parsing phase(syntactic analyser)determines syntactic structure



# Grammars

## Context-Free Grammars

- Developed by Noam Chomsky in the mid-1950s.
- Language generators, meant to describe the syntax of natural languages.
- Define a class of languages called *context-free languages*.

## Backus Normal/Naur Form (1959)

- Invented by John Backus to describe Algol 58 and refined by Peter Naur for Algol 60.
- BNF is equivalent to context-free grammars





NOAM CHOMSKY,
MIT Institute Professor;
Professor of Linguistics,
Linguistic Theory,
Syntax, Semantics,
Philosophy of Language

- **Chomsky and Backus independently came up with essentially equivalent formalisms for specifying the syntax of a language.**
- **Backus focused on a practical way of specifying an artificial language, like Algol.**
- **Chomsky made fundamental contributions to mathematical linguistics and was motivated by the study of human languages.**

Six participants in the 1960 Algol conference in Paris. This was taken at the 1974 ACM conference on the history of programming languages. Top: John McCarthy, Fritz Bauer, Joe Wegstein. Bottom: *John Backus*, *Peter Naur*, Alan Perlis.




# BNF (continued)

A *metalanguage* is a language used to describe another language.

In BNF, *abstractions* are used to represent classes of syntactic structures -- they act like syntactic variables (also called *nonterminal symbols*), e.g.

```
<while_stmt> ::= while <logic_expr> do <stmt>
```

This is a *rule*; it describes the structure of a while statement




# BNF

- A rule has a left-hand side (LHS) which is a **single** non-terminal symbol and a right-hand side (RHS), one or more *terminal* or *nonterminal* symbols.
- A *grammar* is a finite, nonempty set of rules
- A non-terminal symbol is “defined” by its rules.
- Multiple rules can be combined with the | symbol (read as “or”)
- These two rules:

```
<stmts> ::= <stmt>
<stmts> ::= <stmnt> ; <stmnts>
```

are equivalent to this one:

```
<stmts> ::= <stmt> | <stmnt> ; <stmnts>
```

## Non-terminals, pre-terminals & terminals

- A non-terminal symbol is any symbol that is in the RHS of a rule. These represent abstractions in the language (e.g., *if-then-else-statement* in

```
if-then-else-statement ::= if <test>
  then <statement> else <statement>
```

- A terminal symbol is any symbol that is not on the LHS of a rule. AKA *lexemes*. These are the literal symbols that will appear in a program (e.g., *if*, *then*, *else* in rules above).
- A pre-terminal symbol is one that appears as a LHS of rule(s), but in every case, the RHSs consist of single terminal symbol, e.g., <digit> in

```
<digit> ::= 0 | 1 | 2 | 3 … 7 | 8 | 9
```



## BNF

- Repetition is done with recursion
- E.g., Syntactic lists are described in BNF using recursion
- An <ident_list> is a sequence of one or more <ident>s separated by commas.

```
<ident_list> ::= <ident> |
                 <ident> , <ident_list>
```



## BNF Example

Here is an example of a simple grammar for a subset of English.

A sentence is noun phrase and verb phrase followed by a period.

```
<sentence> ::= <nounPhrase> <verbPhrase> .
<nounPhrase> ::= <article> <noun>
<article> ::= a | the
<noun> ::= man | apple | worm | penguin
<verbPhrase> ::= <verb>|<verb><nounPhrase>
<verb> ::= eats | throws | sees | is
```



## Derivations

- A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence consisting of just all terminal symbols.
- It demonstrates, or proves that the derived sentence is "generated" by the grammar and is thus in the language that the grammar defines.
- As an example, consider our baby English grammar

```
<sentence>    ::= <nounPhrase><verbPhrase>.
<nounPhrase> ::= <article><noun>
<article>     ::= a | the
<noun>        ::= man | apple | worm | penguin
<verbPhrase> ::= <verb> | <verb><nounPhrase>
<verb>        ::= eats | throws | sees | is
```

## Derivation using BNF

Here is a derivation for “the man eats the apple.”

<sentence> -> <nounPhrase><verbPhrase>.
<article><noun><verbPhrase>.
the<noun><verbPhrase>.
the man <verbPhrase>.
the man <verb><nounPhrase>.
the man eats <nounPhrase>.
the man eats <article> < noun>.
the man eats the <noun>.
the man eats the apple.




## Derivation

Every string of symbols in the derivation is a *sentential form.*

A *sentence* is a sentential form that has only terminal symbols.

A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded.

A derivation may be either leftmost or rightmost or something else.




## Another BNF Example

```
<program> -> <stmts>
<stmts> -> <stmt>
        | <stmt> ; <stmts>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term> | <term> - <term>
<term> -> <var> | const
```

*Note: There is some variation in notation for BNF grammars. Here we are using -> in the rules instead of ::=.*

Here is a derivation:

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```




## Finite and Infinite languages

- A simple language may have a finite number of sentences.
- An finite language is the set of strings representing integers between -10**6 and +10**6
- A finite language can be defined by enumerating the sentences, but using a grammar might be much easier.
- Most interesting languages have an infinite number of sentences.

## Is English a finite or infinite language?

- Assume we have a finite set of words
- Consider adding rules like the following to the previous example

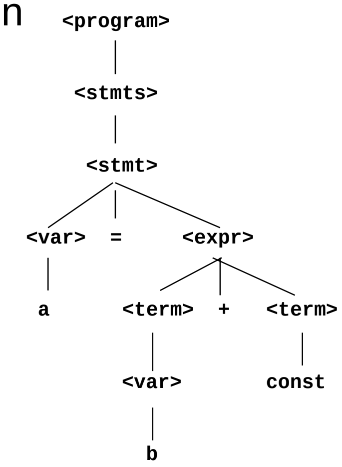
```
<sentence> ::=
 <sentence><conj><sentence>.
<conj>     ::= and | or | because
```

- Hint: Whenever you see recursion in a BNF it's a sign that the language is infinite.
  - When might it not be?




## Parse Tree

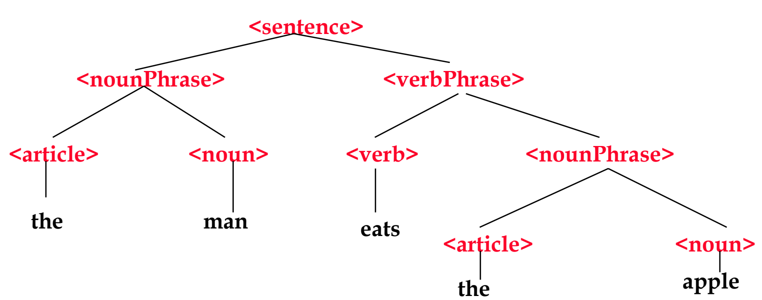
A *parse tree* is a hierarchical representation of a derivation







## Another Parse Tree







## Grammar

A grammar is *ambiguous* iff it generates a sentential form that has two or more distinct parse trees.

Ambiguous grammars are, in general, very undesirable in formal languages.

We can eliminate ambiguity by revising the grammar.

## An ambiguous grammar

Here is a simple grammar for expressions that is ambiguous

```
<expr> -> <expr> <op> <expr>
<expr> -> int
<op> -> +|-|*|/
```

The sentence *1+2*3* can lead to two different parse trees corresponding to *1+(2*3)* and *(1+2)*3*




## Operators

- The traditional operator notation introduces many problems.
- Operators are used in
  - Prefix notation: E.g. Expression *(* (+ 1 3) 2)* in Lisp
  - Infix notation: E.g. Expression *(1 + 3) * 2* in Java
  - Postfix notation: E.g. Increment *foo++* in C
- Operators can have 1 or more operands
  - Increment in C is a one-operand operator: *foo++*
  - Subtraction in C is a two-operand operator: *foo - bar*
  - Conditional expression in C is a three-operand operators: *(foo == 3 ? 0 : 1)*




## Operator notation

- So, how do we interpret expressions like

  (a) 2 + 3 + 4

  (b) 2 + 3 * 4
- While you might argue that it doesn't matter for (a), it can for different operators *(2 ** 3 ** 4)* or when the limits of representation are hit (e.g., round off in numbers, e.g., 1+1+1+1+1+1+1+1+1+1+1+10**6)
- Concepts:
  - Explaining rules in terms of operator precedence and associativity.
  - Realizing the rules in grammars.




## Operators: Precedence and Associativity

- **Precedence** and **associativity** deal with the evaluation order within expressions
- *Precedence* rules specify the order in which operators of different precedence level are evaluated, e.g.:

  "*" Has a higher precedence that "+", so "*" groups *more tightly* than "+"
- What is the results of 4 * 5 ** 6 ?
- A language's precedence hierarchy should match our intuitions, but the result's not always perfect, as in this Pascal example:

  *if A<B and C<D then (*ouch*)*
- Pascal's relational operators have the lowest precedence!

# Operator Precedence: Precedence Table

| Fortran | Pascal | C | Ada |
|---|---|---|---|
| | | ++, -- (post-inc., dec.) | |
| ** | not | ++, -- (pre-inc., dec.), +, - (unary), & (address of), * (contents of), ! (logical not), ~ (bit-wise not) | abs (absolute value), not, ** |
| *, / | *, /, div, mod, and | * (binary), /, % (modulo division) | *, /, mod, rem |
| +, - | +, - (unary and binary), or | +, - (binary) | +, - (unary) |
| | | <<, >> (left and right bit shift) | +, - (binary), & (concatenation) |
| .eq., .ne., .lt., .le., .gt., .ge. (comparisons) | | <, >, <=, >= (inequality tests) | =, /=, <=, >, >= (comparisons) |
| .not. | | ==, != (equality tests) | |




# Operator Precedence: Precedence Table

| | | | |
|---|---|---|---|
| | | & (bit-wise and) | |
| | | ^ (bit-wise exclusive or) | |
| | | \| (bit-wise inclusive or) | |
| .and. | | && (logical and) | and, or, xor (logical operators) |
| .or. | | \|\| (logical or) | |
| .eqv., .neqv. (logical comparisons) | | ?: (if...then...else) | |
| | | =, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, \|= (assignment) | |
| | | , (sequencing) | |




# Operators: Associativity

- *Associativity* rules specify the order in which operators of the same precedence level are evaluated
- Operators are typically either left associative or right associative.
- Left associativity is typical for +, - , * and /
- So A + B + C
  - Means: (A + B) + C
  - And not: A + (B + C)
- Does it matter?




# Operators: Associativity

- For + and * it doesn't matter in theory (though it can in practice) but for – and / it matters in theory, too.
- What should A-B-C mean?

  (A – B) – C ≠ A – (B – C)
- What is the results of 2 ** 3 ** 4 ?
  - 2 ** (3 ** 4) = 2 ** 81 = ??
  - (2 ** 3) ** 4 = 8 ** 4 = 256
- Languages diverge on this case:
  - In Fortran, ** associates from right-to-left, as in normally the case for mathematics
  - In Ada, ** doesn't associate; you must write the previous expression as 2 ** (3 ** 4) to obtain the expected answer

## Precedence and associativity in Grammar

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

An unambiguous expression grammar:

```
<expr> -> <expr> - <term>  |  <term>
<term> -> <term> / const  |  const
```

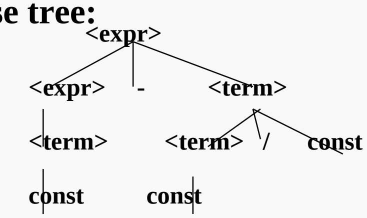



## Precedence and associativity in Grammar

**Sentence:** const – const / const

**Derivation:**
<expr> => <expr> - <term>
=> <term> - <term>
=> const - <term>
=> const - <term> / const
=> const - const / const

**Parse tree:**





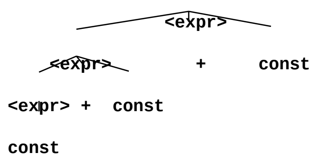


## Grammar (continued)

Operator associativity can also be indicated by a grammar

`<expr> -> <expr> + <expr>  |  const` **(ambiguous)**

`<expr> -> <expr> + const  |  const` **(unambiguous)**







## An Expression Grammar

Here's a grammar to define simple arithmetic expressions over variables and numbers.

Exp ::= num
Exp ::= id
Exp ::= UnOp Exp
Exp := Exp BinOp Exp
Exp ::= '(' Exp ')'

UnOp ::= '+'
UnOp ::= '-'
BinOp ::= '+' | '-' | '*' | '/

*Here's another common notation variant where single quotes are used to indicate terminal symbols and unquoted symbols are taken as non-terminals.*

# A derivation

Here's a derivation of a+b*2 using the expression grammar:

```
Exp =>                 // Exp ::= Exp BinOp Exp
 Exp BinOp Exp =>      // Exp ::= id
 id BinOp Exp =>       // BinOp ::= '+'
 id + Exp =>           // Exp ::= Exp BinOp Exp
 id + Exp BinOp Exp => // Exp ::= num
 id + Exp BinOp num => // Exp ::= id
 id + id BinOp num =>  // BinOp ::= '*'
 id + id * num
 a  + b  * 2
```




# A parse tree

A parse tree for a+b*2:

```
            __Exp__
           /   |   \
       Exp   BinOp   Exp
        |      |   /  |    \
   identifier + Exp BinOp Exp
                 |     |   |
            identifier * number
```




# Precedence

- Precedence refers to the order in which operations are evaluated.
- Usual convention: exponents > mult div > add sub.
- So, deal with operations in categories: exponents, mulops, addops.
- Here's a revised grammar that follows these conventions:

```
Exp ::= Exp AddOp Exp
Exp ::= Term
Term ::= Term MulOp Term
Term ::= Factor
Factor ::= '(' + Exp + ')'
Factor ::= num | id
AddOp ::= '+' | '-'
MulOp ::= '*' | '/'
```




# Associativity

- Associativity refers to the order in which 2 of the same operation should be computed
  - 3+4+5 = (3+4)+5, left associative (all BinOps)
  - 3^4^5 = 3^(4^5), right associative
- Conditionals right associate but have a wrinkle: an else clause associates with closest *unmatched if*

  if a then if b then c else d

  = if a then (if b then c else d)

# Adding associativity to the grammar

Adding associativity to the BinOp expression grammar

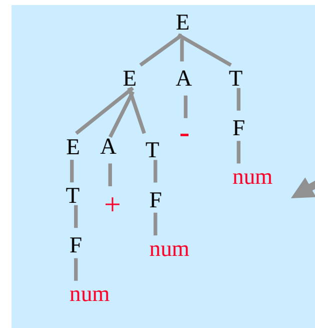
```
Exp    ::= Exp AddOp Term
Exp    ::= Term
Term   ::= Term MulOp Factor
Term   ::= Factor
Factor ::= '(' Exp ')'
Factor ::= num | id
AddOp  ::= '+' | '-'
MulOp  ::= '*' | '/'
```




## Grammar

```
Exp    ::= Exp AddOp Term
Exp    ::= Term
Term   ::= Term MulOp Factor
Term   ::= Factor
Factor ::= '(' Exp ')'
Factor ::= num | id
AddOp  ::= '+' | '-'
MulOp  ::= '*' | '/'
```

## Parse tree




## Derivation

```
Exp =>
Exp AddOp Term =>
Exp AddOp Exp AddOp Term =>
Term AddOp Exp AddOp Term =>
Factor AddOp Exp AddOp Term =>
Num AddOp Exp AddOp Term =>
Num + Exp AddOp Term =>
Num + Factor AddOp Term =>
Num + Num AddOp Term =>
Num + Num - Term =>
Num + Num - Factor =>
Num + Num - Num
```




# Another example: conditionals

- Goal: to create a correct grammar for conditionals.
- It needs to be non-ambiguous and the precedence is else with nearest unmatched if.

```
Statement    ::= Conditional | 'whatever'
Conditional ::= 'if' test 'then' Statement 'else' Statement
Conditional ::= 'if' test 'then' Statement
```

- The grammar is ambiguous. The 1st Conditional allows unmatched 'if's to be Conditionals.

  if test then (if test then whatever else whatever) = correct

  if test then (if test then whatever) else whatever = incorrect

- The final unambiguous grammar.

```
Statement ::= Matched | Unmatched
Matched ::= 'if' test 'then' Matched 'else' Matched
             | 'whatever'
Unmatched ::= 'if' test 'then' Statement
             | 'if' test 'then' Matched 'else' Unmatched
```




# Extended BNF

*Syntactic sugar:* doesn't extend the expressive power of the formalism, but does make it easier to use (more readable and more writable).

Optional parts are placed in brackets ([])

<proc_call> -> ident [ ( <expr_list>)]

Put alternative parts of RHSs in parentheses and separate them with vertical bars

<term> -> <term> (+ | -) const

Put repetitions (0 or more) in braces ({})

<ident> -> letter {letter | digit}

## BNF vs EBNF

BNF:

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

EBNF:

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

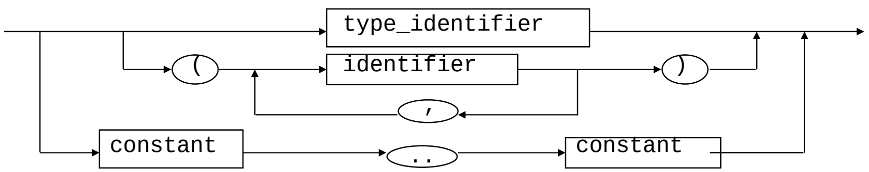


## Syntax Graphs

*Syntax Graphs* - Put the terminals in circles or ellipses and put the nonterminals in rectangles; connect with lines with arrowheads

e.g., Pascal type declarations

Provides an intuitive, graphical notation.






## Parsing

- A grammar describes the strings of tokens that are syntactically legal in a PL
- A *recogniser* simply accepts or rejects strings.
- A generator produces sentences in the language described by the grammar
- A *parser* construct a derivation or parse tree for a sentence (if possible)
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A *recursive descent parser* is a way to implement a top-down parser that is particularly simple.



## Parsing complexity

- How hard is the parsing task?
- Parsing an arbitrary Context Free Grammar is $O(n^3)$, e.g., it can take time proportional the cube of the number of symbols in the input. This is bad!
- If we constrain the grammar somewhat, we can always parse in linear time. This is good!
- Linear-time parsing
  - LL parsers
    - »Recognize LL grammar
    - »Use a top-down strategy
  - LR parsers
    - »Recognize LR grammar
    - »Use a bottom-up strategy

- **LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.**
- **LR(n) : Left to right, Right derivation, look ahead at most n symbols.**

# Recursive Decent Parsing

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all sentential forms that the nonterminal can generate
- The recursive descent parsing subprograms are built directly from the grammar rules
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars (why not?)




- **Basic containment relationship**
  - **All CFGs can be recognized by LR parser**
  - **Only a subset of all the CFGs can be recognized by LL parsers**





# Recursive Decent Parsing Example

Example: For the grammar:

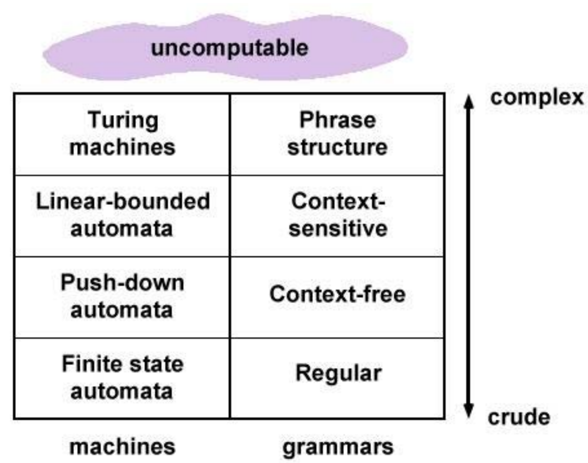
```
<term> -> <factor> {(*|/)<factor>}
```

We could use the following recursive descent parsing subprogram (this one is written in C)

```
void term() {
  factor();     /* parse first factor*/
  while (next_token == ast_code ||
         next_token == slash_code) {
    lexical();  /* get next token */
    factor();   /* parse next factor */
  }
}
```




# The Chomsky hierarchy




- The Chomsky hierarchy has four types of languages and their associated grammars and machines.
- They form a strict hierarchy; that is, regular languages < context-free languages < context-sensitive languages < recursively enumerable languages.
- The syntax of computer languages are usually describable by regular or context free languages.

## Summary

- The syntax of a programming language is usually defined using BNF or a context free grammar
- In addition to defining what programs are syntactically legal, a grammar also encodes meaningful or useful abstractions (e.g., block of statements)
- Typical syntactic notions like operator precedence, associativity, sequences, optional statements, etc. can be encoded in grammars
- A parser is based on a grammar and takes an input string, does a derivation and produces a parse tree.