Programming Languages

Bibhas Adhikari

IIT Kharagpur

September 25, 2020

Basics of Graph theory

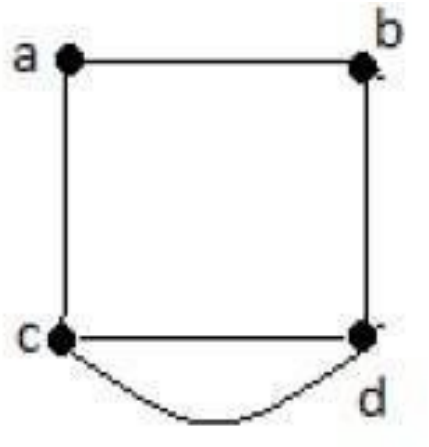


Figure: Example of a multigraph

Degree sequence

If the degrees of all vertices in a graph are arranged in descending or ascending order, then the sequence obtained is known as the degree sequence of the graph. For the undirected simple graph, the vertices $\{d, a, b, c, e\}$, the degree sequence is $\{3, 2, 2, 2, 1\}$

Basics of Graph theory

Path in a graph

A path connecting two vertices a and b in a graph is a sequence of vertices $a = a_1, a_2, \ldots, a_k = b$ such that a_i and $a_{i+1}, i = 1, \ldots, k-1$ are adjacent.

Distance between two vertices

It is number of edges in a shortest path between vertices a and b. If there are multiple paths connecting two vertices, then the shortest path is considered as the distance between the two vertices

- The maximum distance between a vertex to all other vertices is considered as the eccentricity of vertex
- The minimum eccentricity from all the vertices is considered as the radius of the graph
- The maximum eccentricity from all the vertices is considered as the diameter of the graph

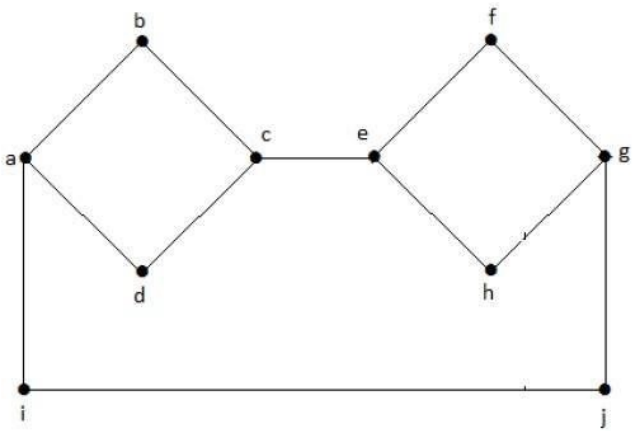
Basics of Graph theory

Connected graph

A graph is said to be connected if there exists a path between every pair of vertices, otherwise it is called disconnected. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

Cycle

A cycle in a graph is a path with no vertex repeated in the sequence of vertices that defines the path, and the initial vertex of the path is same as the terminal vertex of the path



Basics of Graph theory

Acyclic graph

A graph without a cycle is called acyclic

Tree and Forest

A connected acyclic graph is called a tree. In other words, a connected graph with no cycles is called a tree. A disconnected acyclic graph is called a forest. In other words, a disjoint collection of trees is called a forest.

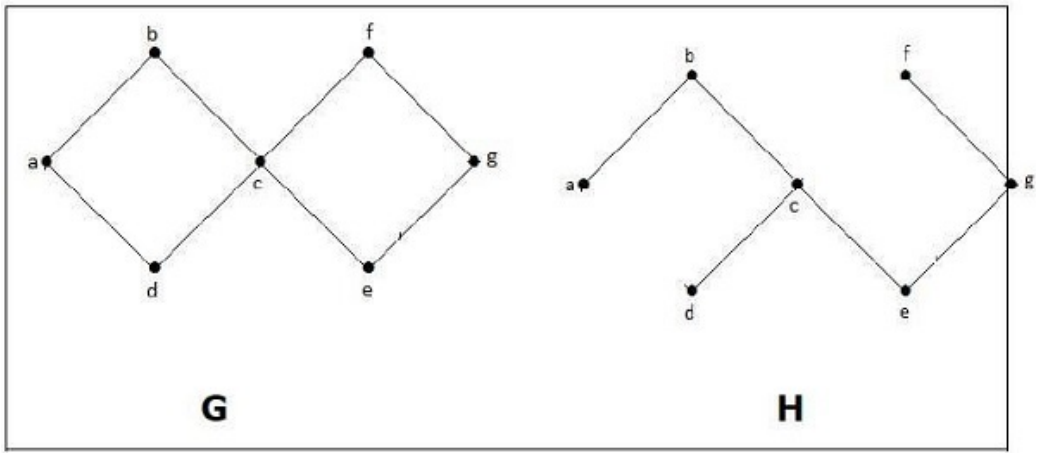


Figure: Example of cycles in a graph

Terminologies in a Tree

Ordered, rooted trees

- A (rooted, ordered) tree is a finite set of vertices/nodes, such that if it is not empty, a particular vertex/node is called the root and the remaining nodes, if they exist, are partitioned between the elements of an (ordered) n-tuple $S_1, S_2, \ldots, \ldots, S_n$, $n \geq 0$, where each S_i , $i \in \{1, \ldots, n\}$ is a tree
- A tree allows us to group nodes into levels where, at level 0, we have the root, at level 1 we have the roots of the trees $S_1, S_2, \ldots, S_n$ and so on.
- The vertex at level 0 is called the root node. If a node n is at level i, and there exists the edge (n, m) then node m is at level $i + 1$
- Given a node, n, the nodes m such that there exists an edge (n, m) are said to be the children of n (and n is said to be their parent); for every node n, a total order is established on the set of all the children of n

Rooted tree

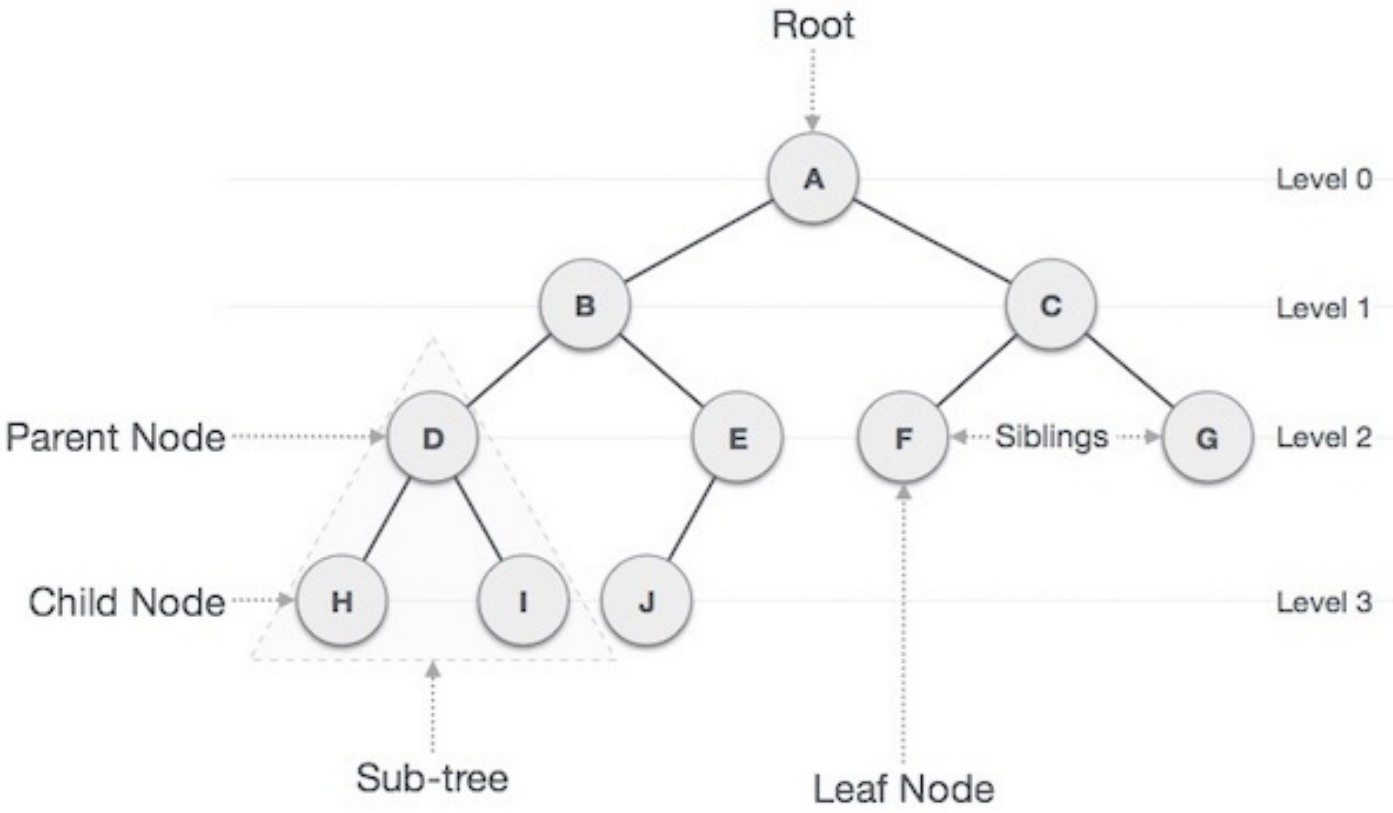


Figure: Example of a rooted tree

Vertices/nodes with the same parent are said to be siblings while nodes without children are said to be leaves. The root is the only node without a parent.

Context-Free Grammars

Example

$G = (\{E, I\}, \{a, b, +, *, -, (,)\}, R, E)$, where R is the following set of productions:

1. E $\rightarrow I$,
2. E$\rightarrow E + E$,
3. E$\rightarrow E * E$
4. E$\rightarrow E - E$,
5. E$\rightarrow -E$,
6. E$\rightarrow (E)$
7. I $\rightarrow a$
8. I $\rightarrow b$
9. I $\rightarrow Ia$
10. I$\rightarrow Ib$

Context-Free Grammars

Example: derivation tree

$$
\begin{aligned}
E &\Rightarrow_3 E * E \\
&\Rightarrow_6 E * (E) \\
&\Rightarrow_2 E * (E + E) \\
&\Rightarrow_1 E * (E + I) \\
&\Rightarrow_8 E * (E + \mathbf{b}) \\
&\Rightarrow_1 E * (I + \mathbf{b}) \\
&\Rightarrow_7 E * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_1 I * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_{10} I\mathbf{b} * (\mathbf{a} + \mathbf{b}) \\
&\Rightarrow_7 \mathbf{ab} * (\mathbf{a} + \mathbf{b})
\end{aligned}
$$

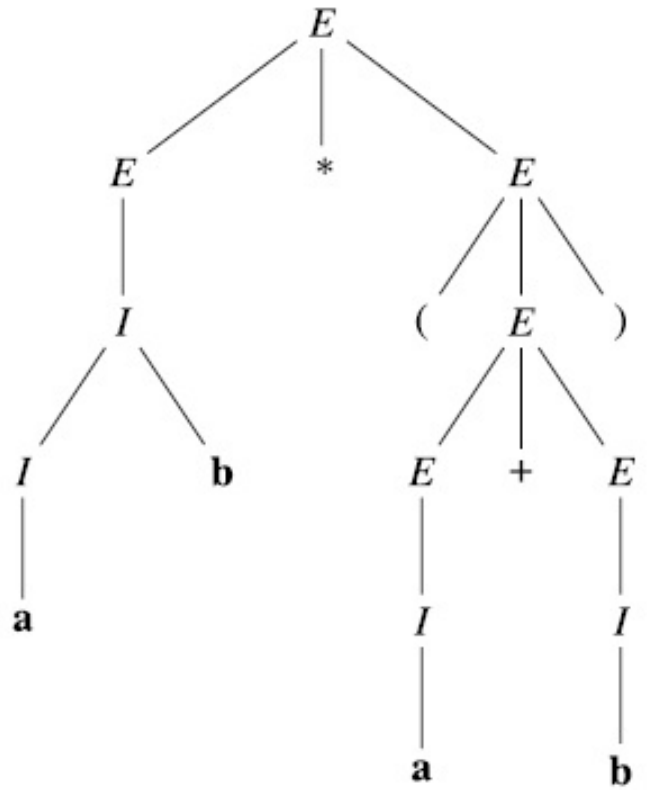


Figure: Derivation and its corresponding tree of $ab * (a + b)$

Context-Free Grammars

Definition: derivation tree

For a grammar $G = (NT, T, R, S)$ derivation tree is an ordered tree in which:

- Each node is labelled with a symbol in $NT \cup T \cup \epsilon$
- The root is labelled with S
- Each interior node is labelled with a symbol in NT
- If a certain node has the label $A \in NT$ and its children are $m_1, \ldots, m_k$ labelled respectively with $X_1, \ldots, X_k$, where $X_i \in NT \cup T$ for all $i \in [1, k]$, then $A \rightarrow X_1 \ldots X_k$ is a production of R
- If a node has label ϵ, then that node is the unique child. If A is its parent, $A \rightarrow \epsilon$ is a production in R

Context-Free Grammars

Example: derivation tree

$$
\begin{aligned}
E &\Rightarrow_2 E + E \\
&\Rightarrow_3 E * E + E \\
&\Rightarrow_1 I * E + E \\
&\Rightarrow_7 \mathbf{a} * E + E \\
&\Rightarrow_1 \mathbf{a} * I + E \\
&\Rightarrow_8 \mathbf{a} * \mathbf{b} + E \\
&\Rightarrow_1 \mathbf{a} * \mathbf{b} + I \\
&\Rightarrow_7 \mathbf{a} * \mathbf{b} + \mathbf{a}
\end{aligned}
$$

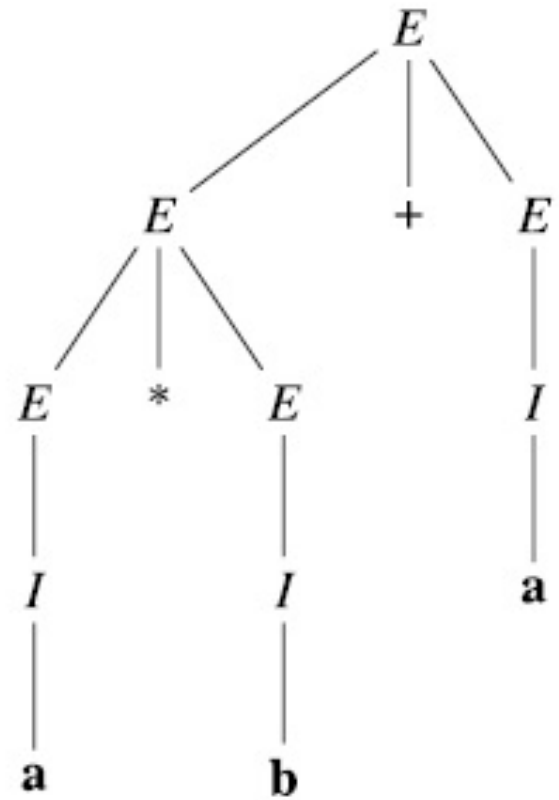


Figure: Derivation and its corresponding tree of $a * b + a$

Context-Free Grammars

Example: derivation tree

$$
\begin{aligned}
E &\Rightarrow_3 E * E \\
&\Rightarrow_1 I * E \\
&\Rightarrow_7 \mathbf{a} * E \\
&\Rightarrow_2 \mathbf{a} * E + E \\
&\Rightarrow_1 \mathbf{a} * I + E \\
&\Rightarrow_8 \mathbf{a} * \mathbf{b} + E \\
&\Rightarrow_1 \mathbf{a} * \mathbf{b} + I \\
&\Rightarrow_7 \mathbf{a} * \mathbf{b} + \mathbf{a}
\end{aligned}
$$

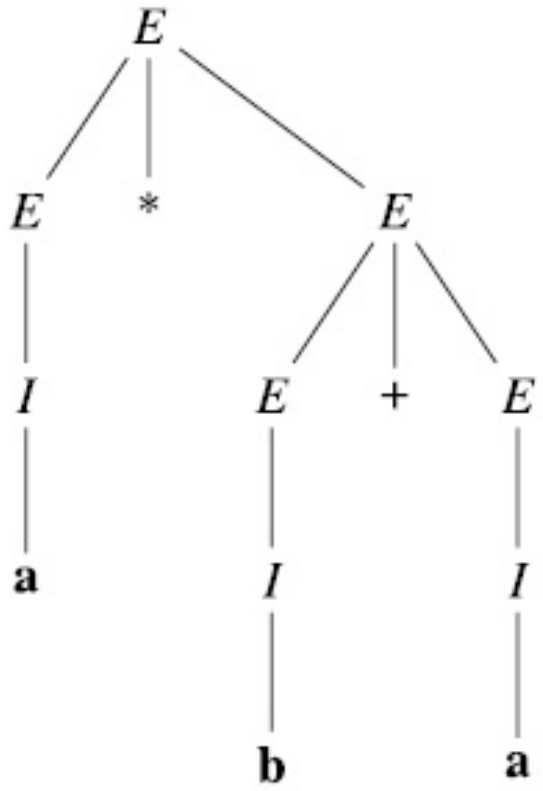


Figure: Another Derivation tree for $a * b + a$

Context-Free Grammars

Ambiguity

- Two different trees producing the same string (see above example)
- Thus the context free grammar is incapable of assigning a unique structure to the string in question

Definition: Ambiguity

A grammar, G, is said to be ambiguous if there exists at least one string of $\mathbb{L}(G)$ which admits more than one derivation tree

However, there are techniques to transform an ambiguous grammar to an unambiguous grammar which can generate the same language

Context-free Grammars

Backus-Naur Form (BNF)

- The arrow "$\rightarrow$" is replaced by "::="
- The non-terminal symbols are written between angle brackets
- Productions with the same head are grouped into a single block using vertical bar ("|") to separate the productions

For instance the productions for E in the above example can be written as:

$$\langle E \rangle ::= \langle I \rangle | \langle E \rangle + \langle E \rangle | \langle E \rangle * \langle E \rangle | \langle E \rangle - \langle E \rangle | - \langle E \rangle | (\langle E \rangle)$$

when $E \rightarrow I$, $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow E - E$, $E \rightarrow -E$, $E \rightarrow (E)$

Context-free Grammar

Example of an unambiguous grammar

$G = (\{E, T, A, I\}, \{a, b, +, *, -, (,)\}, R', E)$, where R' is the following set of productions:

$$E \to T \mid T + E \mid T - E$$
$$T \to A \mid A * T$$
$$A \to I \mid -A \mid (E)$$
$$I \to a \mid b \mid Ia \mid Ib$$

Grammar gives structure to a program (just like a natural language)

- Unary minus ("$-$") has the highest precedence, followed by $*$, followed in their turn by $+$ and binary $-$ (which have the same precedence)
- grammar interprets a sequence of operators at the same level of precedence by association to the right

Context-free Grammar

Statement ::= . . . | *IfThenStatement* | *IfThenElseStatement* |
StatementWithoutTrailingSubstatement
StatementWithoutTrailingSubstatement ::= . . . | *Block* | *EmptyStatement* |
ReturnStatement
StatementNoShortIf ::= . . . | *StatementWithoutTrailingSubstatement* |
IfThenElseStatementNoShortIf
IfThenStatement ::= `if` (*Expression*) *Statement*
IfThenElseStatement ::=
`if` (*Expression*) *StatementNoShortIf* `else` *Statement*
IfThenElseStatementNoShortIf ::=
`if` (*Expression*) *StatementNoShortIf* `else` *StatementNoShortIf*

Figure: Grammar for Java conditional commands

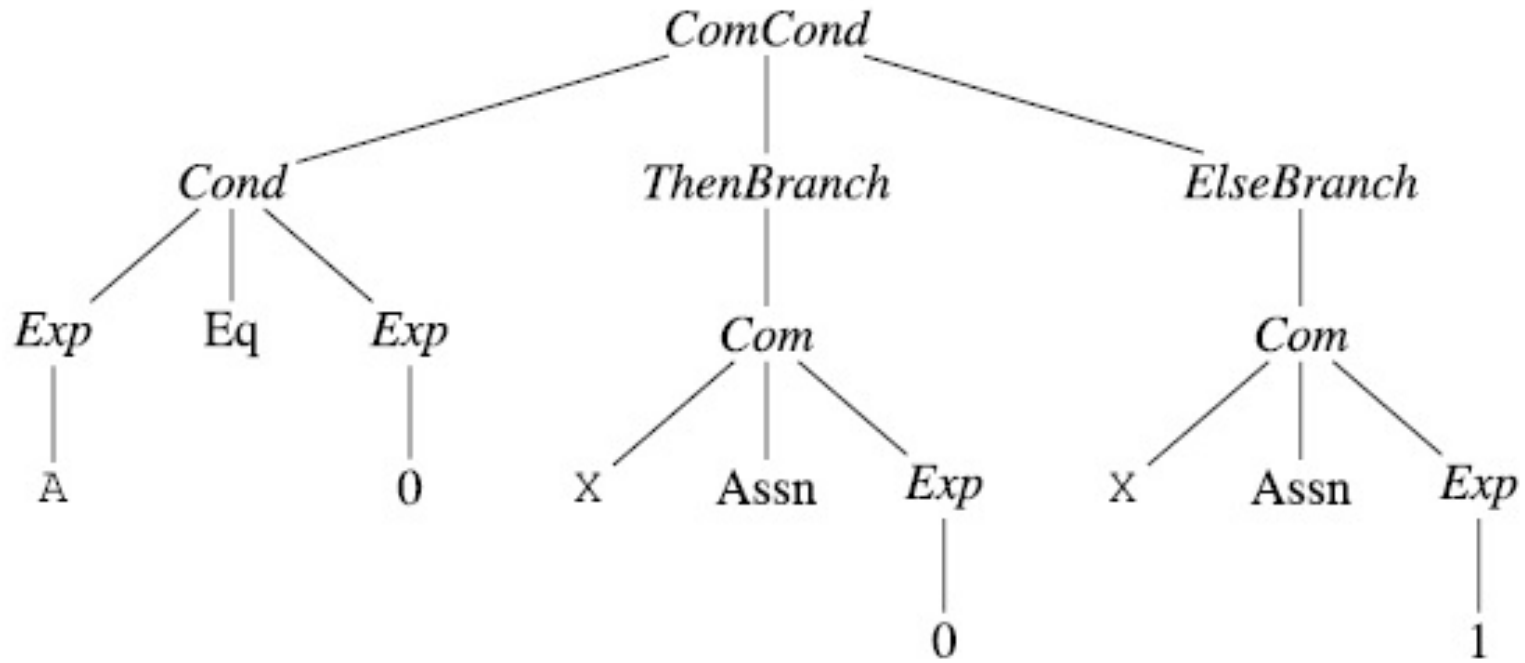
Example of a derivation tree

Pascal string

$if\ A = 0\ then\ X := 0\ else\ X := 1$

Java string

$if\ (A == 0)X = 0;\ else\ X = 1;$



The syntactic correctness of a phrase

- Strings which are correct according to the grammar, can be legal only in a given context
  - In C, the number of actual parameters to a function must be the same as the formal parameters
  - In C, the type of an expression must be compatible with that of the variable to which it is assigned

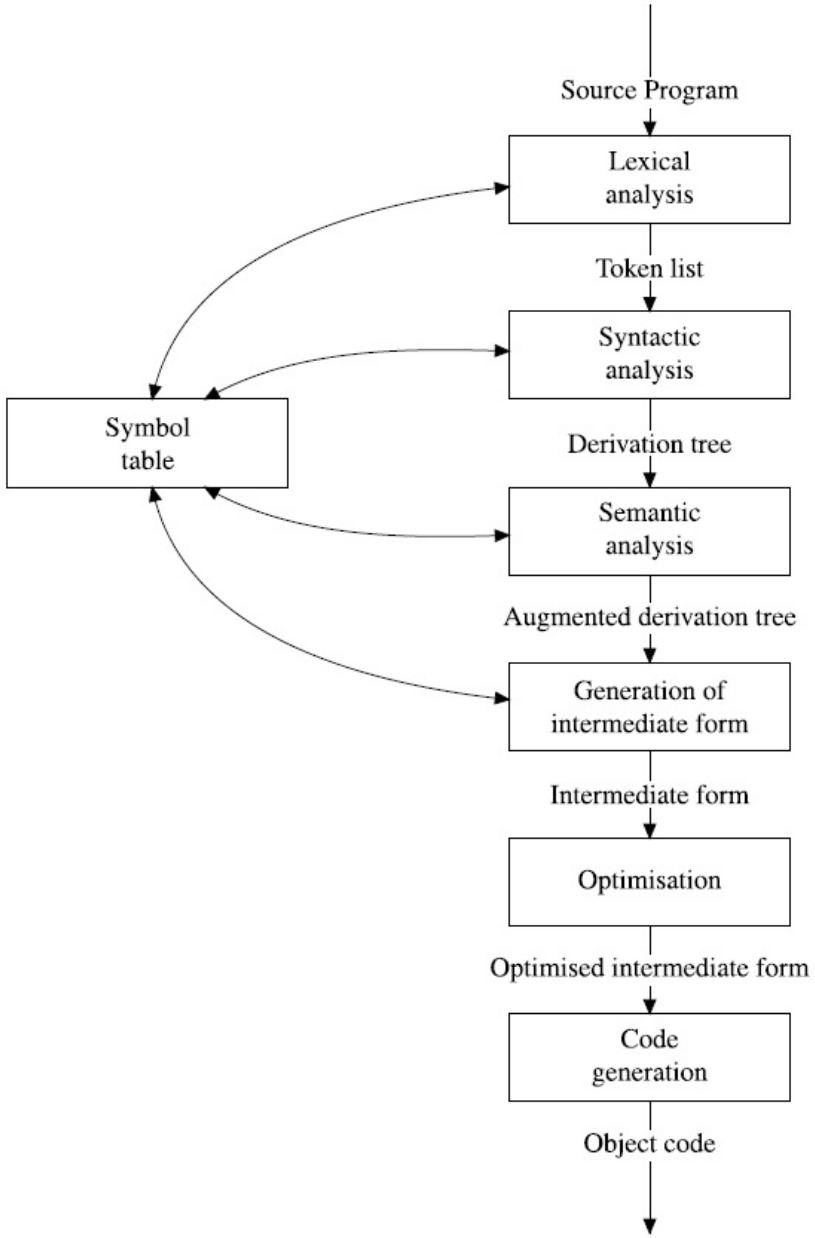
Example

- $x = y + 2$ may be a legal string
- If the language requires the declaration of variables, it is necessary for programs to contain the declarations of x and y before the assignment

- Syntactic constraints - contextual grammars
  - A program is syntactically incorrect when a division by 0 can happen

```
int x, y;
read(x);
y = 10/x;
```

Compiler: how it translates, the logical structure



Organization of a compiler

Lexical Analysis

- The aim of lexical analysis is to read the symbols (characters) forming the program sequentially from the input and to group these symbols into meaningful logical units, which are called *tokens* For example, the string `x = 1 + foo++;` will produce 7 tokens
- Lexical analysis is a particular class of generative grammars (regular grammars)

Regular grammar

- A grammar is said to be regular if all productions are of the form $A \rightarrow bB$ where A and B are non-terminal symbols (B can also be absent or coincide with A) and b is a single terminal symbol
- The subgrammar with initial non terminal symbol I

Syntactic Analysis

- The syntactic analyser (or parser) seeks to construct a derivation tree for this list of tokens. Each leaf of this tree must correspond to a token from the list obtained by the scanner (the above step). Moreover, these leaves, read from left to right, must form a correct phrase (or a sequence of terminal symbols) in the language

Organization of a compiler

Semantic analysis

The derivation tree is subjected to checks of the language's various context-based constraints.

Generation of intermediate forms

This is designed to be independent of both the source and the object languages

Code optimisation

This is designed to be independent of both the source and the object languages

- Removal of useless code
- In-line expansion of function calls.
- Subexpression factorisation
- Loop optimisations