FINAL REPORT

COMPUTER SYSTEMS ENGINEERING
7AM

Clavel Vergara Kimbberly Nicole	171080181
González River Emmanuel Alejandro	171080089
Diez Bonilla Guerrero Dilan Eduardo	171080167
Lopez Tello Rodriguez Mario Ivan	171080102

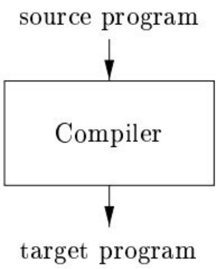
ACTIVITIES WEEK 3 (OCT 5-9, 2020)

INTRODUCTION

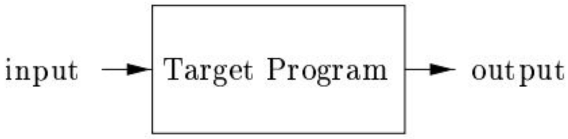
The programming languages are notations of the calculations of people and machines. All software that computers run depends on programming languages because it was written in some language. But before a program can be run, it must first be translated into a format that a computer can execute.

1.1 LANGUAGE PROCESSORS

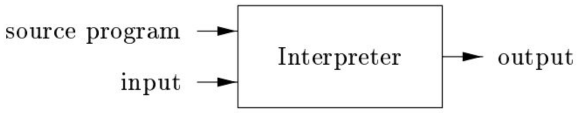
A compiler is a program that can read the source language and translate it into a target language.



If the target program is a machine executable program, then the user can run it to process the inputs and produce outputs.



An interpreter is another type of language processor. Instead of producing a target program as a translation, the interpreter gives us the appearance of directly executing the operations specified in the source program with the inputs provided by the user.



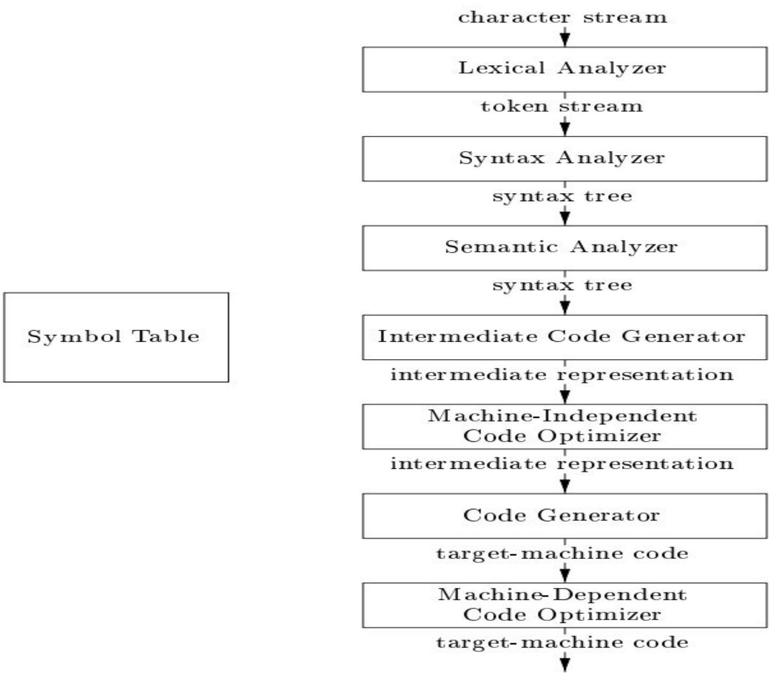
The machine language target program produced by a compiler is faster than an interpreter in assigning inputs to outputs. But usually the interpreter can provide better error diagnostics than a compiler, because it runs the source program instruction by instruction.

1.2 THE STRUCTURE OF A COMPILER

There are two processes in this assignment: analysis and synthesis.

The analysis part divides the source program into components and imposes a grammatical structure on them. Then it uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is poorly formed in terms of syntax, or that it does not have consistent semantics, then it must provide informative messages so that the user can correct it. It also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part builds the desired target program from the intermediate representation and the information in the symbol table. The analysis part is commonly called the front-end of the compiler; the synthesis part (the translation itself) is the back-end



1.2.1 LEXICAL ANALYSIS

The first phase of a compiler is called lexical analysis or scanning. The lexicon analyzer reads the flow of characters that make up the source program and groups them into meaningful sequences, known as lexemes. For each lexeme, the lexicon analyzer produces a form token as output:

name-token, value-attribute

That goes to the next phase, the analysis of the syntax. In the token, the first name-token component is an abstract symbol that is used during the syntax analysis, and the second value-attribute component points to an entry in the symbol table for this token. The information from the symbol table entry is needed for semantic analysis and code generation.

1.2.2 SYNTAX ANALYSIS

The second phase of the compiler is the syntactic analysis or parsing. The parser uses the first components of the tokens produced by the lexicon analyzer to create an intermediate tree-like representation that describes the grammatical structure of the token flow. A typical representation is the syntax tree, in which each inner node represents an operation and the children of the node represent the arguments of the operation.

This tree shows the order in which operations should be carried out in the next assignment:

position = start + speed * 60

1.2.4 INTERMEDIATE CODE GENERATION

After the syntactic and semantic analysis of the source program, many compilers generate an explicit low level, or an intermediate representation similar to machine code, that we can consider as a program for an abstract machine. This intermediate representation must have two important properties: it must be easy to produce and easy to translate in the target machine.

1.2.5 CODE OPTIMIZATION

The machine independent code optimization phase tries to improve the intermediate code, so that a better target code is produced. Generally, better means faster, but other goals can be achieved, such as a shorter code, or a target code that consumes less power.

1.2.6 CODE GENERATION

The code generator receives as input an intermediate representation of the source program and assigns it to the target language. If the target language is machine code, registers or memory locations (locations) are selected for each of the variables used by the program.

1.2.7 MANAGEMENT OF THE SYMBOL TABLE

The symbol table is a data structure that contains one record for each variable name, with fields for the name attributes.

1.2.9 COMPILER CONSTRUCTION TOOLS

Any software developer contains tools such as language editors, debuggers, version managers, profilers, secure testing environments, etc. These tools use specialized languages to specify and implement specific components, and many use quite sophisticated algorithms.

1.4 THE SCIENCE OF BUILDING A COMPILER

A compiler must accept all source programs according to the language specification; the set of source programs is infinite and any program can be very long, possibly consisting of millions of lines of code. Any transformation the compiler makes while translating a source program must preserve the meaning of the program being compiled.

1.4.1 MODELING IN THE DESIGN AND IMPLEMENTATION OF COMPILERS

These models are useful to describe the lexical units of the programs (keywords, identifiers and so on) and to describe the algorithms used by the compiler to recognize those units.

1.4.2 THE SCIENCE OF CODE OPTIMIZATION

Compiler design optimization refers to attempts by a compiler to produce code that is more efficient than obvious code.

Language processors. An integrated software development environment includes many different types of language processors, such as compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.

Compiler phases. A compiler operates as a sequence of phases, each of which transforms the source program from one intermediate representation to another.

Machine and assembler languages. Machine languages were the programming languages of the first generation, followed by assembly languages. Programming in these languages was time consuming and error prone.

Modeling in compiler design. Compiler design is one of the phases in which theory has had the greatest impact on practice. Among the models that have been found useful are: automata, grammars, regular expressions, trees and many others.

Code optimization. Although code cannot truly be "optimized", this science of improving code efficiency is both complex and very important. It constitutes a big part of the compilation study.

High level languages. As time goes by, programming languages take over more and more of the tasks that were previously left to the programmer, such as memory management, type consistency checking, or parallel code execution.

Compilers and computer architecture. Compiler technology influences computer architecture as well as it is influenced by advances in architecture. Many modern innovations in architecture depend on the ability of compilers to extract from source programs the opportunities to effectively use the capabilities of the hardware.

Software productivity and security. The same technology that allows compilers to optimize code can be used for a variety of program analysis tasks, ranging from detecting common program errors to finding that a program is vulnerable to one of the many types of intrusions that hackers have discovered.

Scope rules. The scope of a statement of x is the context in which the uses of x refer to this statement. A language uses the static scope or lexical scope if it is possible to determine the scope of a statement just by analyzing the program. In any other case, the language uses a dynamic scope.

Environments. The association of names with memory locations and then with values can be described in terms of environments, which assign the names to the memory locations, and states, which assign the locations to their values.

Block structure. It is said that languages that allow nesting blocks have block structure. An x name in a nested block B is within the scope of a declaration D of x in a surrounding block, if no other declaration of x exists in an intermediate block.

Parameter step. Parameters are passed from a procedure that makes the call to the procedure that is called, either by value or by reference. When large objects are passed by value, the values that are passed are actually references to the same objects, which results in an effective reference call.

Use of aliases. When parameters are (effectively) passed by reference, two formal parameters can refer to the same object. This possibility allows a change in one variable to change the other.

ACTIVITIES WEEK 4 (OCT 12-16, 2020)

VIDEO 1

A compiler is an example of how theory translates into practice.

The complexity of a compiler arises from the fact that it is required to map a program as requirements that are written in high-level languages (CA program).

Lattice theory is used to develop static analyses (it is based on linear algebra and parallelization).

Cache analysis uses static analysis and probability theory.

The use of software testing requires a data flow analysis and estimation approach.

A compiler takes a cleanup and knows the source program (this can be in c or c plus).

The output is an assembler program for the particular machine.

The assembler output is called relocatable machine code that is combined with library files.

A lexical parser takes how to enter a character flow.

The lexical analyzer takes a source program as input and is then known as a token sequence.

Analyzers can be automatically generated from regular expression specifications.

A lexical analyzer is a finite-state deterministic automaton.

An analyzer corresponds to a thrust automaton.

The syntax analyzer looks at the tokens and finds out if the syntax is in accordance with a grammatical specification.

Analyzers cannot handle context-sensitive features of programming languages.

A parameter cannot be detected by the parser.

The syntax tree is produced by a syntax parser.

The parser also shows information in the symbol table or in the syntax tree.

The machine code generator needs to know the types of variables to generate the appropriate types of instructions.

The static semantics of programming languages can be specified using what are the attribute grammars.

Attribute grammars are an extension of context-free grammars.

The type of intermediate code is based on the application.

The program dependency graph is useful in automatic parallelization, instruction programming, software piping, etc.

The dependency graph shows the dependency between various types of declarations in the program.

The best ones can be by time space or energy consumption.

VIDEO 2

The common feature of languages is that there are four character sequences.

The basic lexical analysis in finite-state automata

The lecture analyzer makes both the lexical and the analyzer more efficient.

Tokens are patterns and lexemes are strings of characters.

A float is a reserved word.

A string of characters that logically belongs to worlds is a token.

Tokens are treated as terminal symbols of the grammar that specifies a language.

A pattern is the set of strings for which the same token is produced.

A lexeme is the sequence of characters paired by a pattern to form the corresponding token.

The lexical token analyzer cannot detect any significant errors except very simple ones.

The errors are called by the syntactic analyzers.

Transition diagrams are finite-state variants (automata) used to implement regular definitions and to recognize tokens.

Transition diagrams model lexical analyzers.

A string is a finite sequence of juxtaposed symbols which are symbols placed one after the other.

Language is defined as a set of strings of symbols from some alphabet.

Sigma star are all possible strings on the particular alphabet each subset of sigma star is a language.

Languages are represented by regular expressions.

The hierarchy of languages is known as Chomsky's hierarchy based on respecting the inventor who proposed this hierarchy.

Delimited linear automata accept context-sensitive languages and can be specified using context-sensitive grammars.

Delta tells how the machine progresses from one state to another by consuming a certain input symbol.

Delta is a transition function.

The language accepted by a finite state automaton is the set of all strings given by it.

ACTIVITIES WEEK 5 (OCT 19-23, 2020)

VIDEO 1

A deterministic finite automaton has exactly one transition in each symbol or state.

There can be more than one transition in a symbol or carrier state.

It is possible to make a transmission to a set of states of each NFA state.

Each subset of the NFA states is a possibility of one state.

All DFA states that contain some final state as a member would be the final DFA states.

Epsilon in NFA Empty string island as we know it.

Epsilon is a silent transition and does not consume any input.

A particular NFA converted as well as the same language that is in an epsilon.

Regular expressions are specific.

Regular expressions help specify lexical analyzers.

The initial state of a DFA can be represented by q0

Each subset of an NFA is possibly a DFA state

The symbol "*" is called as Kleene's lock

To show all the strings that 0 and 1 can create, the Kleene's lock is occupied as follows (0+1)*

It is an NFA if there is a sequence corresponding to a string, starting from an initial state to a final state.

VIDEO 2

- Transition diagrams are generalized deterministic finite automata.
- The edges of transition diagrams can be labeled with a regular definition.
- Some acceptance states can be indicated as retraction states.

- The transition diagram must be translated manually into a lexical analysis program.
- The transition diagrams should be tested and then all matches recorded and the longest match should be able to sort the transition diagrams properly.
- Using Lex to generate lexical analyzers makes it easy for the compiler writer.
- Lex has a language for describing expressions, which is at the heart of lexical analysis.
- The Lex I enrolled knows all the regular expressions, specifications for each of the patterns to be detected in lexical analysis.
- Lex generates a pattern comparator for the regular expression.
- The Lex tool programs that are appropriate for lexical analysis.
- The general structure of a Lex program is divided into rules and user subroutines, which are essential parts and specifications of Lex.
- The Lex program section contains regular definitions.
- The general structure of Lex:
- {Definitions}
- %%
- {Rules}
- %%
- {User subroutine}
- Definitions and subroutines are optional. The second %% is optional but the first %% indicates the beginning of the rules
- The regular expressions correspond to the grammars of 3 types of Chomsky's hierarchy. Regular expressions are a way of specifying patterns. A pattern is a way of describing a string of characters.
- The language that is recognized by these regular expressions (r) , are called language generated by the regular expression L(r) .
- The basic operations of a regular expression are represented as
- It is called by | meaning that it can be a or b. This operation is equivalent to the union, since both a and b would be valid as pattern lexemes.

- a | b selection between a and b.

Concatenation:

It is constructed by putting a symbol next to the other and does not use any character to represent it, therefore the equivalent lexeme has to be "ab".

ab concatenation of a and b

Kleen lock:

Denoted by the character * identifies a concatenation of symbols including the empty string, i.e. from 0 to more instances.

a* indicates the Kleen lock.

VIDEO 3

Context-free grammars are the basis for the specification of programming languages.

Analyzing languages in context is based on pushdown automata.

Regular language recognition is based on finite-state automata.

There are two types of analysis: one is top-down analysis, and the other is bottom-up analysis.

Top-down analysis studies LL and recursive descent, syntactic analysis techniques.

The bottom-up analysis, we will study the techniques of LR analysis, as well as, a tool called yacc that is based on LR analysis.

A grammar can be written to describe the structural syntax of well-formed programs or correct programs.

The rules of grammar establish how functions are made from the list of parameters and declarations.

The rules of grammar will tell you how statements are made up of expressions, and in turn, how expressions are made up of numbers, names, parentheses, and so on.

The yacc tool is used to generate analyzers automatically.

Yacc is a tool that takes a grammar specification and writes a syntactic C-analyzer that recognizes valid sentences for that grammar.

Without context, grammars are a subclass of programming languages.

There are different types of languages and grammars, there are regular, contextless, languages, context-sensitive and type-0 languages.

Context-free grammars are used to specify context-free languages and are the most useful for programming.

A free context grammar is denoted as G.

G=(N,T,P,S)

Where:

N: It is a finite set of what is known as non-terminal or variable.

T: It is a finite set of what is known as terminals.

S: Is a non-terminal this is a start symbol.

P: It is a finite set of productions.

The analyzer verifies that the token string for a program in that particular programming where the language can be generated from the grammar we have provided as a basis for the analyzer.

In the PDA system the symbols mean:

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the alphabet of the pile
- q0 ∈ Q is the initial state
- Z0 ∈ Γ is the battery symbol
- F ⊆ Q is the set of final states
- δ is the transition function

Referral Tree

The derivation is represented by ==>

It is represented by x ==> y y is the application of a production rule a→b to a string x to convert it into another string or word y

It is a representation used to describe the process of derivation of a sentence, where the following is fulfilled:

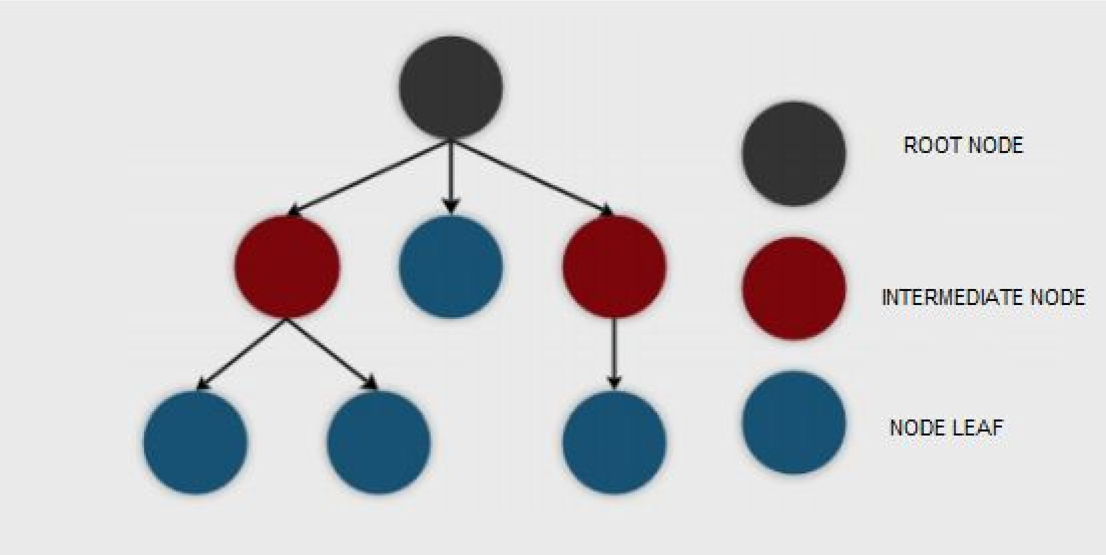
The root of the tree is the initial symbol of the grammar.

The intermediate nodes of the tree are the non-terminal ones of the productions used. These intermediate nodes will have as many children as the right side of the production has elements.

The leaf nodes will be the terminals.

Left and right derivations

Derivations can be made starting from the leftmost non-terminal symbol in the string, and these derivations are called left derivations, or substitutions can be made in the initial string starting from the rightmost non-terminal symbol and these derivations are called right derivations. Depending on which one we choose, we get one result or another:



ACTIVITIES WEEK 6 (OCT 26-30, 2020)

VIDEO 1

A PDA M is a system (Q . ∑ . Γ . 𝛿 . q0. Z0. F), where.

- Q is a finite set of states
- ∑ is the input alphabet or is the stack alphabet.
- qo ϵ Q is the start-up state
- Z0 ϵ Γ the start symbol on the stack (initialization)
- F ϵ Q is the set of final states.
- 𝛿 is the transition function, Q x U x { ϵ } x Γ to finite subsets of Q x Γ*.

A typical input of 𝛿 is given by

𝛿 (q.a.z) = {(p_1 𝛾1) . ((p_2 . 𝛾2) ... (pm . ∩m)}

The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p1, replace symbol z with the string 𝛾i, and advance the input header by one symbol.

- The leftmost symbol of 𝛾i will be the new top of the stack.
- a in the above function 𝛿 could be ϵ, in which case, the input symbol is not used and the input head does not advance.
- For a PDA M, we define L(M), the language accepted by M by final state, is L (M) = {w | (q0 . w . Z0) □* (p . ϵ . 𝛾), for some p ϵ F 𝛾 E Γ'}
- We define N(M), the language accepted by M by empty stack, as N(M) = {w | (q0. w . Z0) □* (p . ϵ . 𝛾), for some p ϵ Q.
- When acceptance is by empty stack, the set of final states is irrelevant and, in general, we set F = φ
- As in the case of NFA and DFA, PDA also has two versions: NPDA and DPDA.
- However, NPDA are strictly more powerful than DPDA.
- For example, the language, L = {wwR | w ϵ {a . b} +} can only be recognized by an NPDA and not by any DPDA.

- At the same time, the language, L = {WWWR | w ϵ {a. b} +}, can be recognized by a DPDA

- In practice we need DPDA, since they have exactly one possible movement at any instant.

- Our analyzers are all DPDA

- Parsing is the process of constructing a parse tree for a sentence generated by a given grammar.

- If there are no restrictions on the language and the form of grammar used, parsers for context-free languages require O(n3) time (where n is the length of the parsed string).

  - Cocke-Younger-Kasami algorithm

  - Earley Algorithm

- Context-free subsets of languages usually require O(n) time.

  - Predictive analytics using LL(1) grammars (top-down analysis method)

  - Shift-Reduce parsing using LR(1) grammars (bottom-up parsing method)

- Top-down analysis using predictive analytics, traces the leftmost derivation in the chain while building the parse tree

- It starts from the start symbol of the grammar and "predicts" the following production used in the derivation

- This "prediction" is aided by analysis tables (built off-line).

- The next output to be used in derivation is determined by using the following input symbol to look up the analysis table (look-ahead symbol)

- The analysis table contains more than one production

- At the time of parsing table construction, if two productions become eligible to be placed in the same parsing table space, the grammar is declared ineligible for predictive parsing.

Let the given grammar G

- The input is expanded with k symbols, $k, k is the grammar advance.

- Introduces a new, non-terminal S' and a production, S' à S$K, where S is the start symbol of the given grammar.
- Consider only the derivations on the left and assume that the grammar has no useless symbols
- A production A à a in G is called a strong LL(k) production, if in G

  S' is* WA γ is W α γ is* wzy

  S' is* w'A δ is w' β δ is* w'zx

  |z| = k . z ϵ ∑*. w and w' ϵ ∑*, then α = β

- A (non-terminal) grammar is strong LL(k) if all its productions are strong LL(k).
- Strong LL(k) grammars do not allow different productions of the same nonterminal to be used even in two different derivations, if the first k symbols of the strings produced by α γ and β δ are the same.
- Example: S to Abc | aAcb, A to ϵ|b|c

  S is a non-terminally strong LL(1)

- S' è S$ è Abc$ è bc$. Bbc$ and cbc$, in application of the productions, A à ϵ, A à b, and A à c, respectively. z = b, b or c, respectively.
- S' è S$ è aAcb$ è acb$. abcb$ and accb$, according to the application of the productions, A à ϵ, A à b, and A à c, respectively. z = a, in all three cases.
- In this case, w = w' = ϵ, α = Abc, β = aAcb, but z is different in the two derivations, in all derivative chains.
- Therefore, the nonterminal S is LL(1).

A is not strong LL(1)

- S' is* Abc$ is bc$, w is ϵ, Z = b, α = ϵ (A à ϵ)

S' is* aAbc$ is bbc$, w' = ϵ, Z = b, β = b (A à b)

- Although the anticipated searches are the same (z = b), a ≠ β, and therefore, the grammar is not strong LL(1).

A is not strong LL(2)

- S' is* Abc$ is bc$, w = ϵ, Z = bc, α = ϵ (A → ϵ)

S' is* aAcb$ is abcb$, w' = a, Z = bc, β = b (A → b)

- Although the lookaheads are the same (z = bc), a ≠ β, and therefore, the grammar is not strong LL(2).

A is strong LL(3) because the six strings (bc$, bbc, cbc, cb$. bcb, ccb) can be distinguished using 3-symbol in advance (details are for home work).

- We call LL(1) strong as LL(1) from now on and we will not consider anticipated searches longer than 1

- The classical condition for the LL(1) property uses sets FIRST and FOLLOW

- If α is any string of grammatical symbols (α ϵ (N U T)"), then FIRST (α) = {a | a ϵ T. and a è* ax. X ϵ T*}

FIRST (ϵ) = {ϵ}

- If A is non-terminal, then

FOLLOW (A) = {a | S is* αAaβ, α. β ϵ (N U T)* . a ϵ T U {$}}

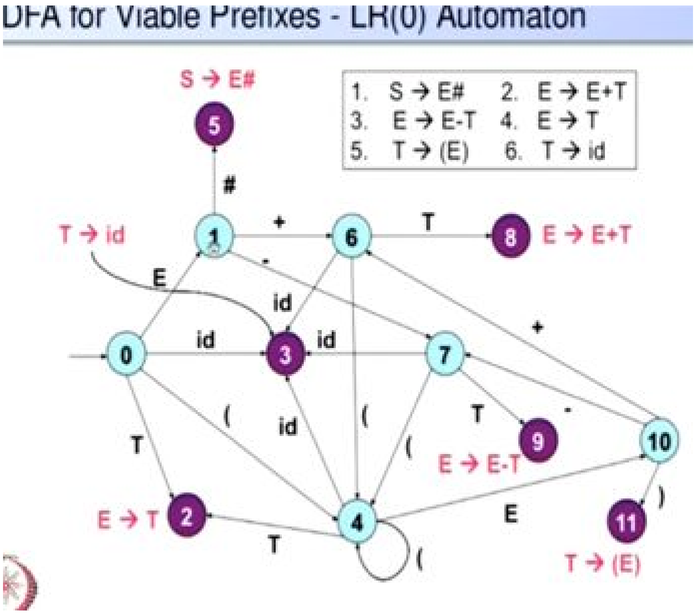
- FIRST (α) is determined by α alone, but FOLLOW (A) is determined by the "context" of A, i.e., the derivations in which A appears.

VIDEO 2

Building an LR 0 automaton

Example

How LR 0 analysis occurs with respect to this particular DFA.

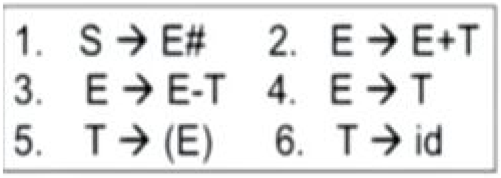


DFA for Viable Prefixes - LR(0) Automaton

1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

Let us consider the simple string identification derived from this grammar. So we will start with S going to E hash, apply this production. And then when we apply the production E going to T and finally we apply the production T, going to id. We get the string id all this is shown in the following image.

This table is a simplification of a whole diagram of an automaton.



1. S → E#	2. E → E+T
3. E → E-T	4. E → T
5. T → (E)	6. T → id

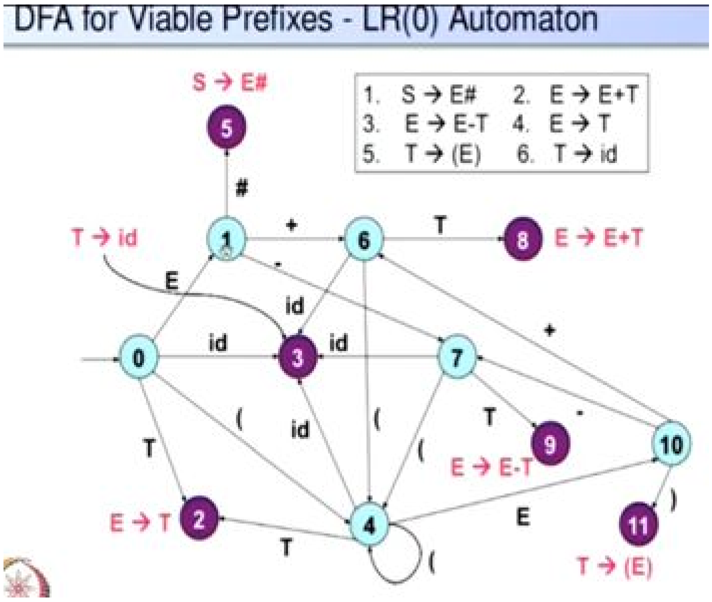
Now we begin to explain how a DFA LR 0 automaton is realized.

We will start with state 0, which is the initial state and then in the input id we go to state number 3, remembering that the initial state is 0, now on the stack id is pushed onto the stack as we see the stack will be the line that helps the zero state to get to state 3, again it is pushed down the stack instead of 3 says it is a reduced state and the reduction is by the production T going to Id, then we go to state number 2 but now we have pushed the non terminal T on the stack and a state number 2 is also on stack now it says in that state reduce from E to t, we take out state 2 and the non terminal T off the stack again we will get state 0 as the top of the stack.

And now since the non-terminal on the left side is E we apply the go to function in state 0 and the non-terminal E which will take us to state number 1, what follows is that the next input symbol which is a hash because it's a change state it's not a reduce state and in the hash goes state 5 where it presents a reduction of S which goes to E hash as indicated in state 5.

Now from state 1 to 6 will take us on the plus path and from state 6 to id and this means that there is a reduction, then remembering that 3 and 6 are on the id stack, then state 6 will take us on the non-terminal T and this will take us to state 8 where we have on the stack E à E+T which indicates a reduction we go on the E path then on the + path and finally on the T path.

This is how an automaton of this style is explained knowing how the paths go from that to which states are going as it is that a state is called when we find a state that contains for example Eà E + T this in a nutshell in the reduction of a path to reach the state where it is.



DFA for Viable Prefixes - LR(0) Automaton

1. S → E# 2. E → E+T
3. E → E-T 4. E → T
5. T → (E) 6. T → id

Change and reduce action

If a state contains an element of the form [Aà alpha] it is called reduce item.

Then a reduction from a to alpha is the action in that state, if there are no "reduce items" in a state, then the appropriate action must be changed.

There could be change-reduce conflicts or reduce-reduce conflicts in a state.

Both change and reduce elements are present in the same state as (S-R conflict) or there is more than one reduced element in a state as (R-R conflict).

It is normal to have more than one shift element in a state if shift conflicts are possible.

If there are no S-R or R-R conflicts in any state of a DFA LR0 then the grammar is LR(0), otherwise it is not LR(0).

How to build the SRL analysis table

You have a stack in which you can store states (non-negative integers) or grammar symbols (terminal or non-terminal). There is also a pair of tables that are named action and go to. The action table contains the actions that can be executed by the automaton.

The actions are specified with a letter that specifies the action and the state number to which the automaton must go after executing it.

They can be:

1. Displacement: it is symbolized by the letter d and a number indicating the state to which the automaton must pass.

Reduction: it is symbolized by the letter r and a number that indicates the number of the rule by which it is reduced.

3. Accept: it is symbolized by the letter a. It does not have a suffix number since it is not necessary; the automaton stops automatically when this command is encountered.

4. Nothing: if an entry in this table does not contain any of the three previous alternatives, then there is a syntax error in the current position. Algorithms used by the Syntax Analyzer Generator 29 The ir_a table contains non-negative integers that are the states to which the automaton has to go after executing a reduction.

VIDEO 3

RECURSIVE DESCENT PARSING

- Top-down analysis strategy
- A function procedure for each non-terminal
- The functions are called recursively, according to the grammar
- They can be automatically generated by the grammar
- Manual coding is also easy
- Error recovery is superior

AUTOMATIC GENERATION OF RD PARSERS

- The scheme is based on the production structure
- The get_token() function gets the following token from the lexical analyzer
- The error() function prints error message

ANÁLISIS DE ABAJO HACIA ARRIBA

- Build small analysis trees and then put these trees together to create a larger one.

- This process is known as reduction

- Shift-reduce analysis is implemented

- The handle concept is used to detect when reductions are to be made.

The parsing algorithm has 4 main functions:

Shift: This function is used to move the next symbol entered to the top of the stack.

Reduce: This function applies a grammar rule to some of the recent parse trees, joining them as a tree with a new symbol.

Accept: This indicates that the parse was successfully completed.

Error: This is responsible for calling the recovery routine.

LR ANALYSIS

- They are generated automatically using the analysis generators

- It is a general shift-reduce method of non-reversal

ACTIVITIES WEEK 7 (NOV 2-6, 2020)

VÍDEO 1

LR parsing is a bottom-up parsing method and is synonymous with left-to-right scanning with the rightmost derivation in reverse.

LR(k). It is left-to-right scanning with rightmost derivation in reverse, where k is the number of anticipated search tokens:

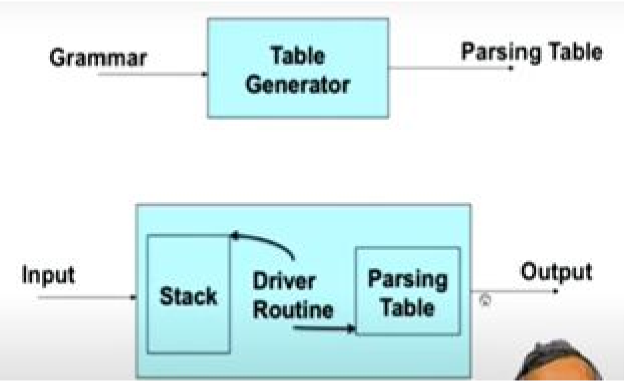
K= 0.1 are of practical interest.

LR parsers are also generated automatically using parser generators.

LR grammars are a subset of CFGS for which LR syntactic parsers can be constructed and are context-free grammars.

The following image is a syntactic parser generator, the generator is a grammar device as input and generates a parser called an LR parser table. The table fits into another box containing a stack and a driver routine, this whole set is the parser.

So here the driver routine and the parse table are stacked together, it takes the program as input and delivers as possible output a free syntax.



LR analyzer configuration.

A configuration of an LR analyzer is:

(S0X1S2X2......XmSm aiai+1....an) where stack unspent input S0,S1.....Sm, are the states of the analyzer, and X1,X2..........Xm are grammatical symbols (terminal or non-terminal).

An initial parser configuration (S0.a1a2...an) where , S0 is the initial state of the parser , and a1a2.....an is the string to be parsed.

The table can obtain two parts of analysis:

1.- The action table can have four types of input: reduction, acceptance or error.

2.- The goto table provides the following state information state information to be used after a reduced movement.

LR grammar.

A grammar is said to be LR(k) if for any given input string, at each step of any rightmost derivation, the manager B can be detected by examining the string and is scanned for at most the first k symbols of the unused input string t.

Let us understand how to construct the deterministic finite state automaton that tracks the states of the parser and tells us when to switch and when to reduce.

A feasible prefix of an enunciative form Bt, where B denotes the identifier is any prefix of O3. A feasible prefix cannot contain symbols to the right of the identifier.

Example:

SàE#, EàE+T|E-T| T,TàID|(E)SàE#EàE+TàE+(E)# àE+(T)# àE+(ID)#.

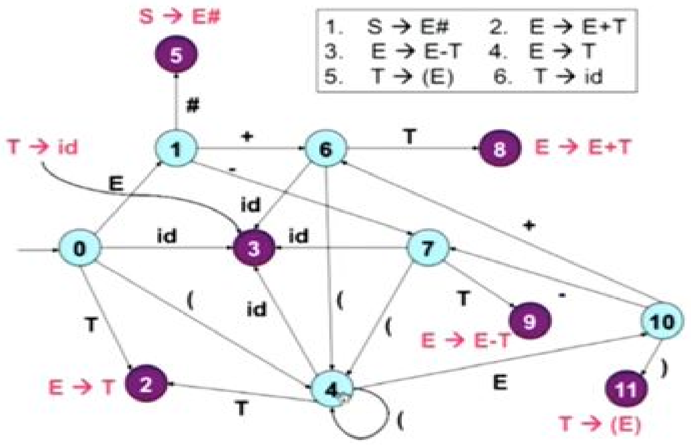
PARSING THEOREM LR

The set of all feasible prefixes of all correct sentence forms of a grammar is a regular language.

The DFA of this regular language can detect identifiers during LR parsing. When the DFA reaches a reduced state, the corresponding viable prefix cannot grow any further and thus indicates a reduction.

The compiler can construct this DFA using the grammar all syntactic parsers have such a DFA built in.

Aquí mostramos un ejemplo de un DFA:



So exactly this there are some states that are in a greenish blue and there are some other states that are in violet, the states that are violet color have a production associated with them and these are the reduction states.

I already know that states 5,8,3,2,9 and 11 are the reduction states and the other states which are 0,1,6,7,10 and 4 which are in a blue-green color are the change states.

One particular DFA as shown in the above picture there is no definite final state in fact all the states are final, so in this DFA from the initial state it does not matter which way it goes.

VIDEO 2

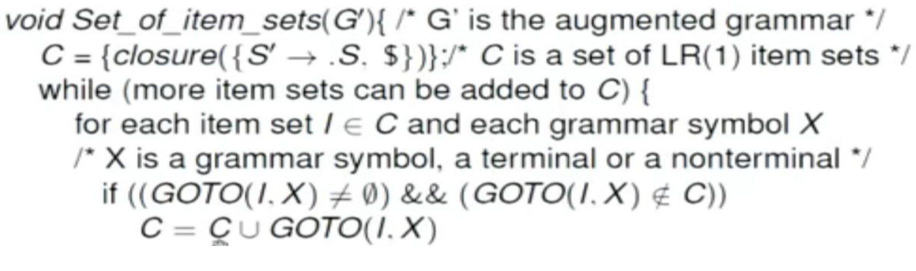
CHANGE AND REDUCE ACTIONS

- If there are no reduced items in a state, change is the appropriate action.
- There can be change-reduce and reduce-reduce conflicts in a state
- It is normal to have more than one item exchanged in a state.
- More than one reduced item must be present in a state
- If there is no SR or RR conflict in any LR DFA state, while the grammar is LR, it is not LR.

If the grammar is not LR, we will try to resolve conflicts in the states by using a look-ahead symbol

SR conflicts can be resolved using the FOLLOW set, the grammar is said to be SLR

All entries not defined in the rules will create an error.



```
void Set_of_item_sets(G'){ /* G' is the augmented grammar */
   C = {closure({$S' \to .S,\ \$$})};/* C is a set of LR(1) item sets */
   while (more item sets can be added to C) {
      for each item set $I \in C$ and each grammar symbol X
      /* X is a grammar symbol, a terminal or a nonterminal */
         if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
            $C = C \cup GOTO(I, X)$
```

Each set in C corresponds to a DFA state.

This is the DFA that recognizes viable prefixes.

VIDEO 3

LALR(1) Analyzer

LR(1) parsers have a large number of states.

For C, many thousands of states.

An SLR(1) (or LR(0) DFA) parser for C will have a few hundred states with a lot of conflict.

LALR(1) parsers have exactly the same number of states as SRL(1) parsers for the same grammar and are derived from SLR(1) parsers for the same grammar and are derived from LR(1) parsers.

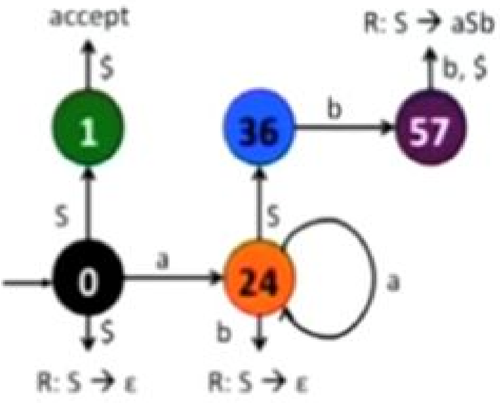
The central part of the LR(1) elements the part following the omission of the look-ahead symbol is the same for several LR(1) states the look-ahead symbols will be different.

Merge the states with the same core together with the symbol anticipatory search and rename them.

The ACTION and GOTO parts of the parser table are modified and merge the rows of the parser table corresponding to the merged states, replace the old names of the states with the corresponding new names for the corresponding new names for the merged states.

For example.

If states 2 and 4 are merged into a new state 24, and states 3 and 6 are merged into a new state 36 all references to states 2,4,3 and 6 will be replaced by 24,24 36 and 36.



ACTIVITIES WEEK 8 (NOV 9-13, 2020)

VIDEO 1

- Semantic coherence that cannot be handled at the parsing stage is handled here

- Parsers cannot handle context-sensitive features of programming languages.

- These are static semantics of the programming languages and can be checked by the semantic parser

- Variables are declared before use

- Types match on both sides of assignments

- Parameter types and numbers match in declaration and usage

- Compilers can only generate code to check the dynamic semantics of programming languages at runtime

- whether an overflow will occur during an arithmetic operation

- whether array bounds will be crossed during execution

- whether recursion will cross stack boundaries

- whether heap memory will be insufficient

Samples of static semantic checks in main

- Types of py dot prod match return type

- The number and type of dot_prod parameters are the same both in its declaration and in use

- p is declared before use, same for ayb

Samples of static semantic checks in dot_prod or d and i are declared before use

the type of d matches the return type of dot_prod

The type of d matches the result type of "*".

The elements of the xey matrices are compatible with

- Let G = (N. T. P. S) be a CFG and let V = NUT.

- Each symbol X of V has a set of attributes (denoted by XaXb, etc.) associated with it.

- Two types of attributes: inherited (denoted by AI (X)) and synthesized (denoted by AS (X)).

- Each attribute takes values from a specific domain (finite or infinite), which is its type.

- Typical domains of attributes are integers, reals, characters, strings, booleans, structures, etc.

- New domains can be constructed from given domains by mathematical operations such as cross product, map, etc.

- matrix. a map, N → D, where, N and D are domains of natural numbers and the given objects, respectively

- structure: a cross product, A, x A2 x ... x An, where n is the number of fields in the structure, and A, is the domain of the i-th field.

VIDEO 2

Attribute grammars.

They are context-free grammars extending symbol of the set N union T has some attributes associated with it.

There are two types of inherited and synthesized attributes.

A synthesized attribute for a non-terminal A at a node N of a syntactic tree is defined by a semantic rule associated with the production in N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of the values of the attributes in the child of N, and in N itself.

An inherited attribute for a non-terminal B at node N of a parse tree is defined by a semantic rule associated with the production in the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of the values of the attributes in the parent of N, in N itself, and in its siblings.

The second class of syntax-driven definitions is known as definitions with inherited attributes. The idea of this class is that, among the attributes associated with the body of a production, the arrows of the dependency graph can go from left to right, but not the other way around (hence they are called "left-inherited attributes"). To put it more precisely, each attribute must be:

1. synthesized.

2. Inherited, but with the rules constrained as follows. Suppose there is a production A → X1X2 --- Xn, and there is an inherited attribute Xi.a, which is computed by a rule associated with this production. Then, the rule can use only:

(a) The inherited attributes associated with header A.

(b) The inherited or synthesized attributes associated with the occurrences of the symbols X1, X2, ..., Xi-1 located to the left of Xi.

An attribute cannot be synthesized and inherited at the same time, but a symbol can have both types of attributes.

Symbol attributes are evaluated in a parse tree by making passes over the parse tree.

(c) Inherited or synthesized attributes associated with this same occurrence of Xi, but only in a form in which there are no cycles in a dependency graph formed by the attributes of this Xi.

The synthesized attributes are computed bottom-up from the leaves upwards, always synthesized from the attribute values of the children of the node.

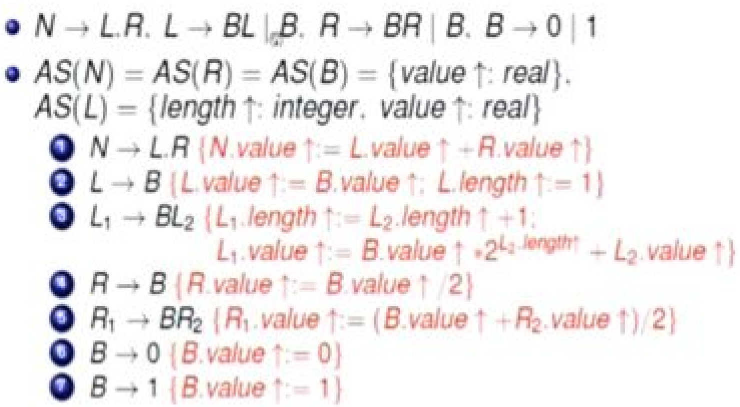
The leaf nodes (terminals) have synthesized attributes (only) initialized by the lexical analyzer and cannot be modified inherited attributes flow from the node in question.

Example

AG for the evaluation of a real number from its bit string representation.

Example: If you have a bit string 110.101=6.625 this will be its decimal value.

The free grammar N goes to L point R, L generates several bits on the left side of the point, and if the grammar L goes to BL or B similarly, R generates bits on the right side of the point and the grammar is R that goes to BR or BB is of course a bit 0 or 1.



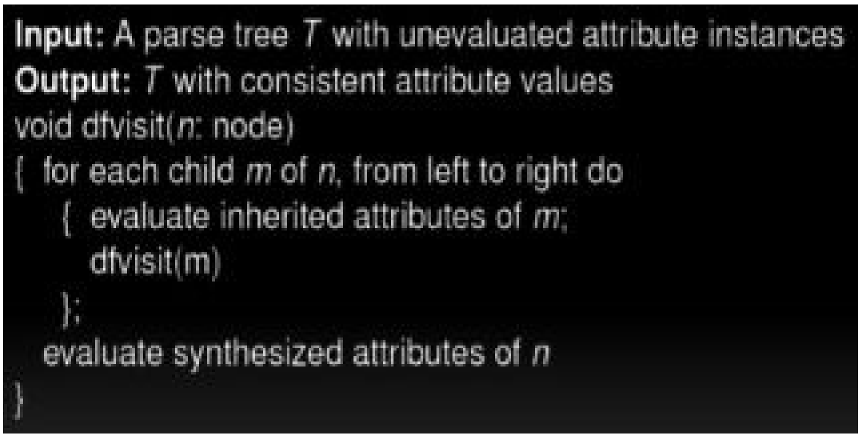
- $N \to L.R.\ L \to BL \mid B.\ R \to BR \mid B.\ B \to 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value\uparrow: real\}.$
  $AS(L) = \{length\uparrow: integer.\ \ value\uparrow: real\}$
  1. $N \to L.R\ \{N.value\uparrow := L.value\uparrow + R.value\uparrow\}$
  2. $L \to B\ \{L.value\uparrow := B.value\uparrow;\ L.length\uparrow := 1\}$
  3. $L_1 \to BL_2\ \{L_1.length\uparrow := L_2.length\uparrow + 1;$
     $L_1.value\uparrow := B.value\uparrow * 2^{L_2.length\uparrow} + L_2.value\uparrow\}$
  4. $R \to B\ \{R.value\uparrow := B.value\uparrow /2\}$
  5. $R_1 \to BR_2\ \{R_1.value\uparrow := (B.value\uparrow + R_2.value\uparrow)/2\}$
  6. $B \to 0\ \{B.value\uparrow := 0\}$
  7. $B \to 1\ \{B.value\uparrow := 1\}$

VIDEO 3

GRAMMAR OF ATTRIBUTES

- Each symbol X or V is associated with a set of attributes
- There are two types of attributes: Inherited and synthesized.
- Each attribute takes values from a specified domain

ATTRIBUTE EVALUATION ALGORITHM FOR LAGS



```
Input: A parse tree T with unevaluated attribute instances
Output: T with consistent attribute values
void dfvisit(n: node)
{  for each child m of n, from left to right do
     {  evaluate inherited attributes of m;
        dfvisit(m)
     };
   evaluate synthesized attributes of n
}
```

GRAMMAR OF ATTRIBUTE TRANSLATION

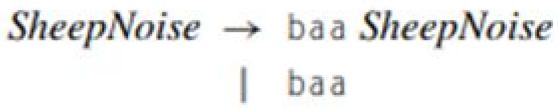
- Apart from computational rules, some programs segment that performance
- As a result of these code segment actions, the order of evaluation can be constrained
- These actions can be added to SAG and LAG.

ACTIVITIES WEEK 11 (30 NOV-4 DIC, 2020)

CONTEXT-FREE GRAMMARS (CONTEXT-FREE GRAMMARS)

To describe programming language syntax, we need a notation more powerful than regular expressions that still leads to efficient recognizers. The traditional solution is to use a context-free grammar (cfg).

A context-free grammar, G, is a set of rules that describe how to form sentences. The collection of sentences that can be derived from G is called a language defined by G, denoted G. The set of languages defined by context-free grammars is called a set of context-free languages. An example may help. Consider the following grammar, which we call SN:



$$
\begin{aligned}
\textit{SheepNoise} \rightarrow\ & \texttt{baa}\ \textit{SheepNoise} \\
|\ \ & \texttt{baa}
\end{aligned}
$$

The first rule or production says "SheepNoise can derive the word baa followed by more SheepNoise".SheepNoise is a syntactic variable that represents the set of strings that can be derived from the grammar. We call such a syntactic variable a non-terminal symbol. Each word in the language defined by the grammar is a terminal symbol. The second rule says "SheepNoise can also derive the string baa".

To derive a sentence, we start with a prototype string containing only the target symbol, SheepNoise. We choose a non-terminal symbol, α, in the prototype string, choose a grammar rule, $\alpha \rightarrow \beta$, and rewrite α with β. We repeat this rewriting process until the prototype string contains no more non-terminals, at which point it is composed entirely of words, or terminal symbols, and is a sentence in the language.

DERIVATION: is a sequence of rewriting steps that begins with the starting symbol of the grammar and ends with a phrase in the language.

SENTENCE FORM: is a string of symbols that occurs as a step in a valid derivation.

CONTEXT-FREE GRAMMAR: For an L language, its CFG defines the sets of symbol strings that are valid sentences in L.

SENTENCE: A string of symbols that can be derived from the rules of a grammar.

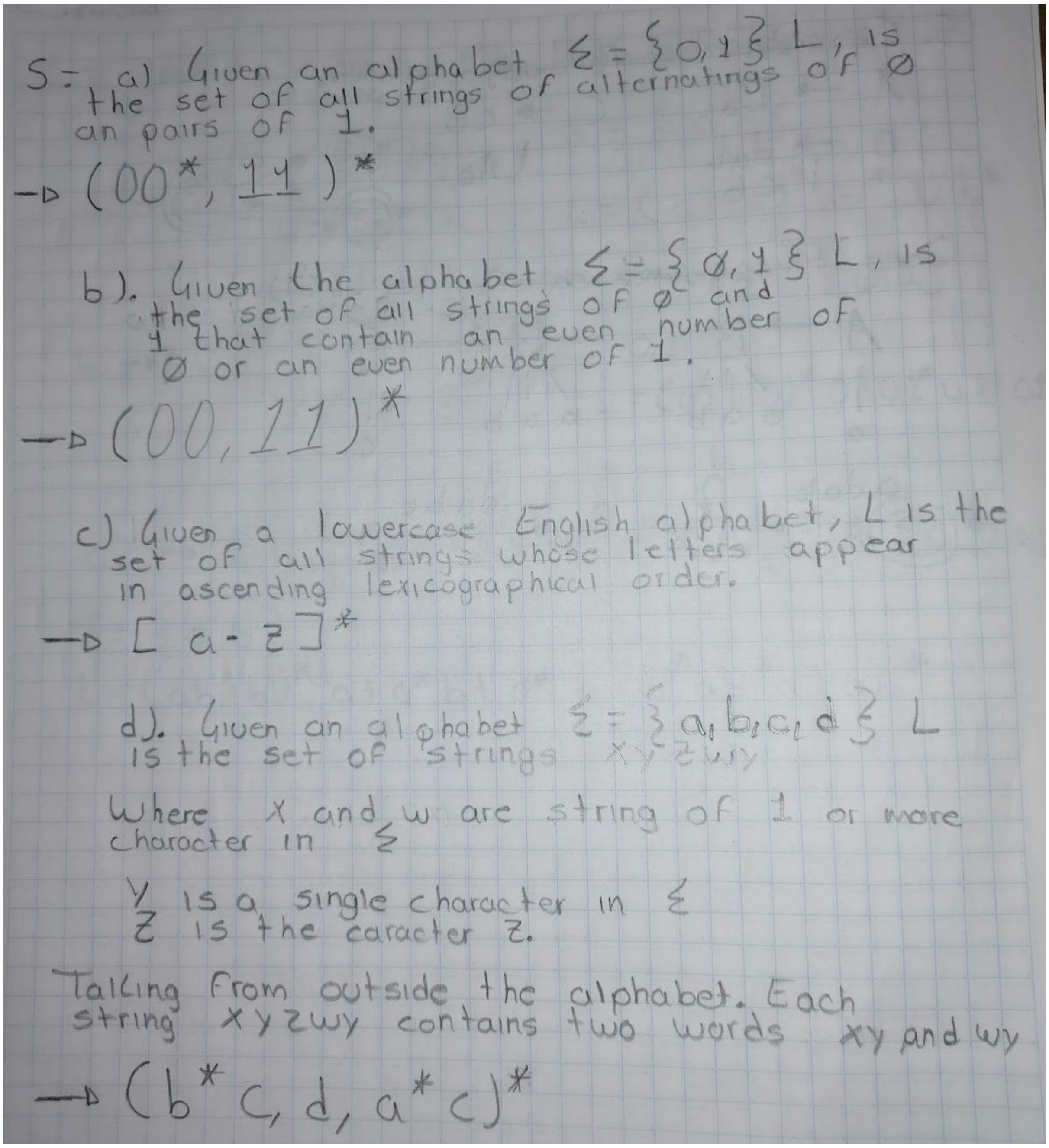
PRODUCTION: Each rule in CFG is called a production.

NON TERMINAL SYMBOL: A syntactic variable used in the productions of a grammar.

TERMINAL SYMBOL: A word that can appear in a sentence.

A word consists of a lexeme and its syntactic category. Words are represented in a grammar by their syntactic category.

EXAMPLES AND EXERCISES



S = a) Given an alphabet $\Sigma = \{0, 1\}$ L, is
the set of all strings of alternatings of Ø
an pairs of 1.

→ $(00^*, \underline{11})^*$

b). Given the alphabet $\Sigma = \{Ø, 1\}$ L, is
the set of all strings of Ø and
1 that contain an even number of
Ø or an even number of 1.

→ $(00, 11)^*$

c) Given a lowercase English alphabet, L is the
set of all strings whose letters appear
in ascending lexicographical order.

→ $[a - z]^*$

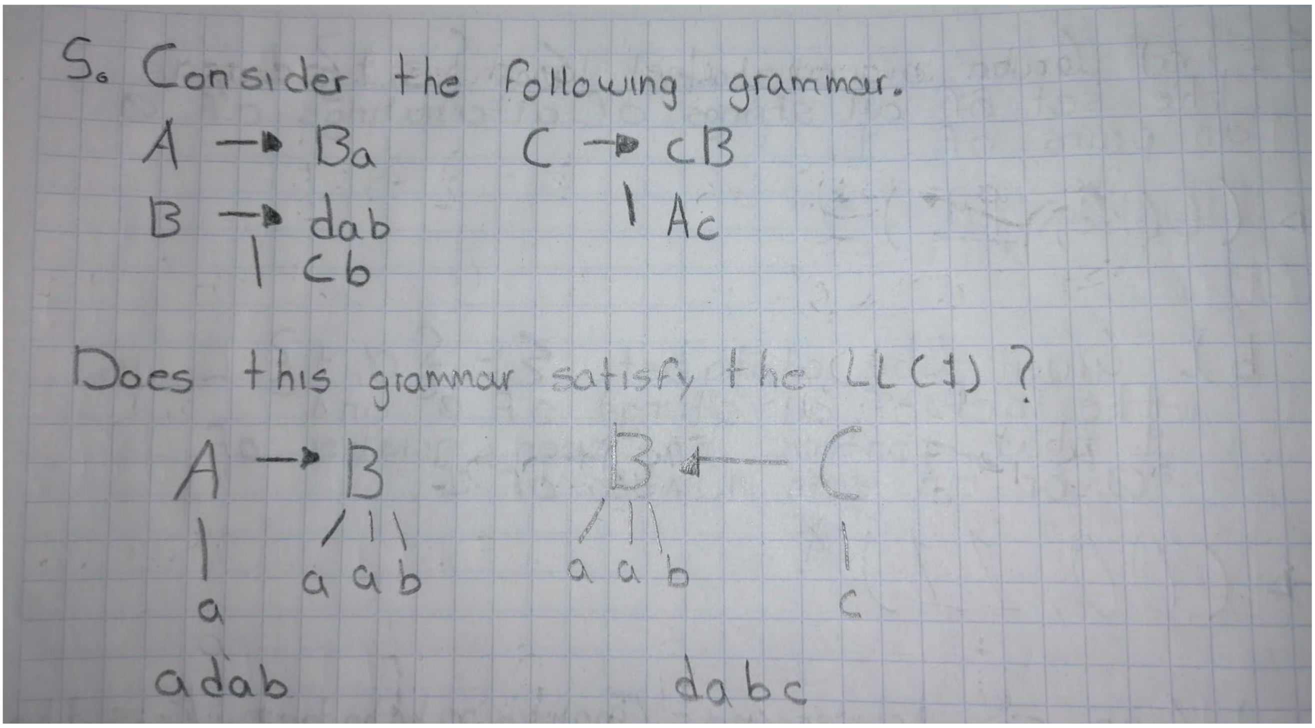
d). Given an alphabet $\Sigma = \{a, b, c, d\}$ L
is the set of strings xyzwy

Where x and w are string of 1 or more
character in Σ

y is a single character in Σ
z is the caracter z.

Talking from outside the alphabet. Each
string xyzwy contains two words xy and wy

→ $(b^* c, d, a^* c)^*$



5. Consider the following grammar.

A → Ba　　　C → cB

B → dab　　　| Ac

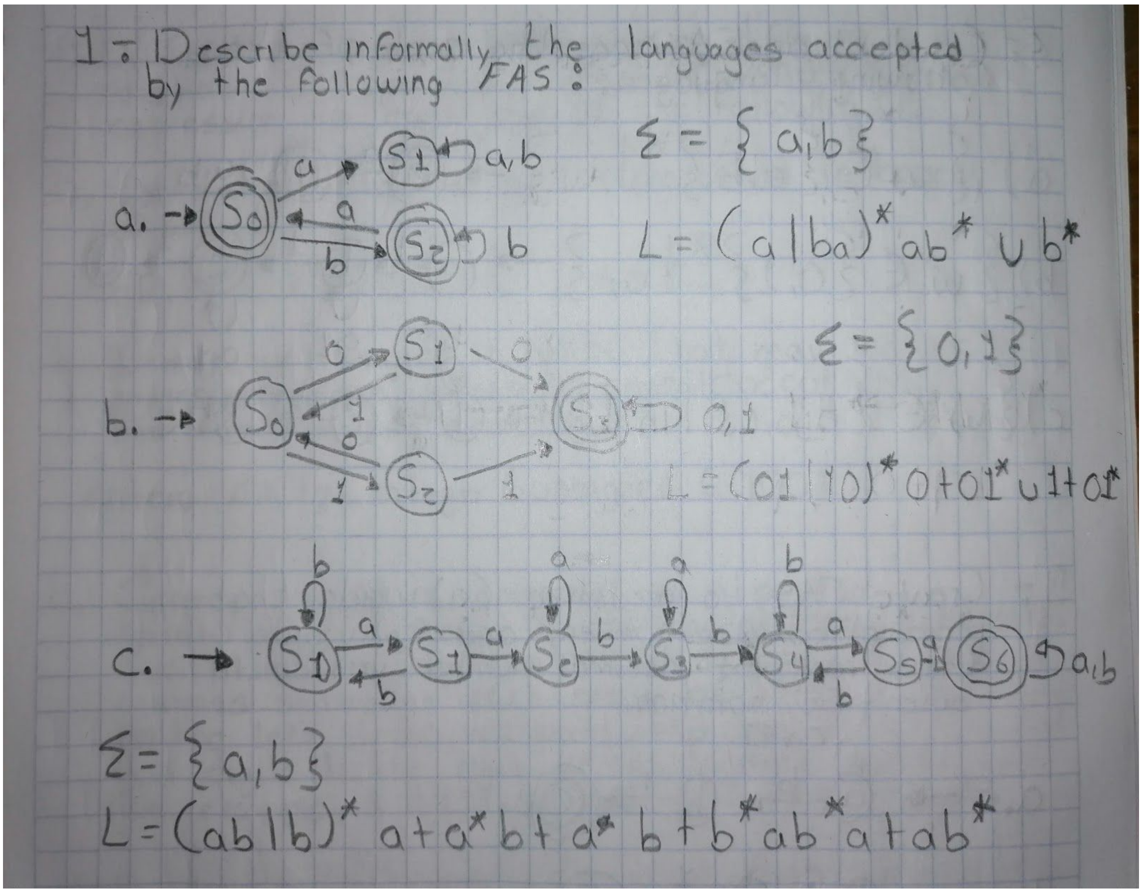
　| Cb

Does this grammar satisfy the LL(1) ?

A → B
a
a a b
B ← C
a a b
c
adab
dabc

ACTIVITIES WEEK 12 (DIC 7-11, 2020)

CHAPTER 2



1.- Describe informally the languages accepted by the following FAS:

a. $\Sigma = \{a, b\}$

$L = (a|ba)^* ab^* \cup b^*$

b. $\Sigma = \{0, 1\}$

$L = (01|10)^* 0+01^* \cup 1+01^*$

c. $\Sigma = \{a, b\}$

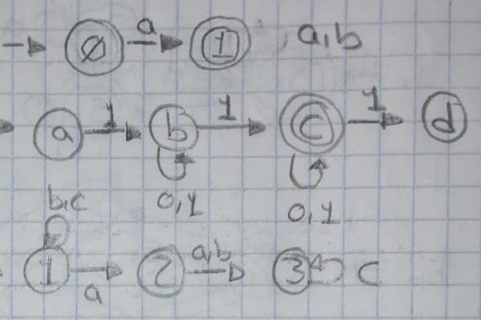
$L = (ab|b)^* a+a^*b+a^* b+b^*ab^*a+ab^*$

2.- Construct an FA accepting each of the following languages:

a) $\{w \in \{a,b\}^* \mid w\}$

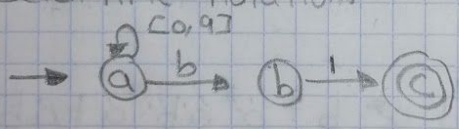
b) $\{w \in \{0,1\}^* \mid w\}$

c) $\{w \in \{a,b,c\} \mid \text{in } w\}$

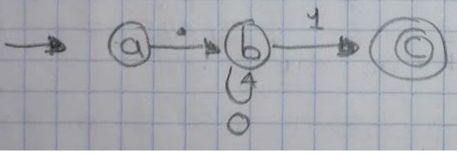


3.- Create FAS to recognize (a) words that represent compler numbers and (b) words that represent decimal number written in scientific notation.

a.



b



4.- Different programming languages use different notations to represent integers. Construct a regular expression for each one of the following:

a. Nonnegative integers in C represented in bases 10 and 16.

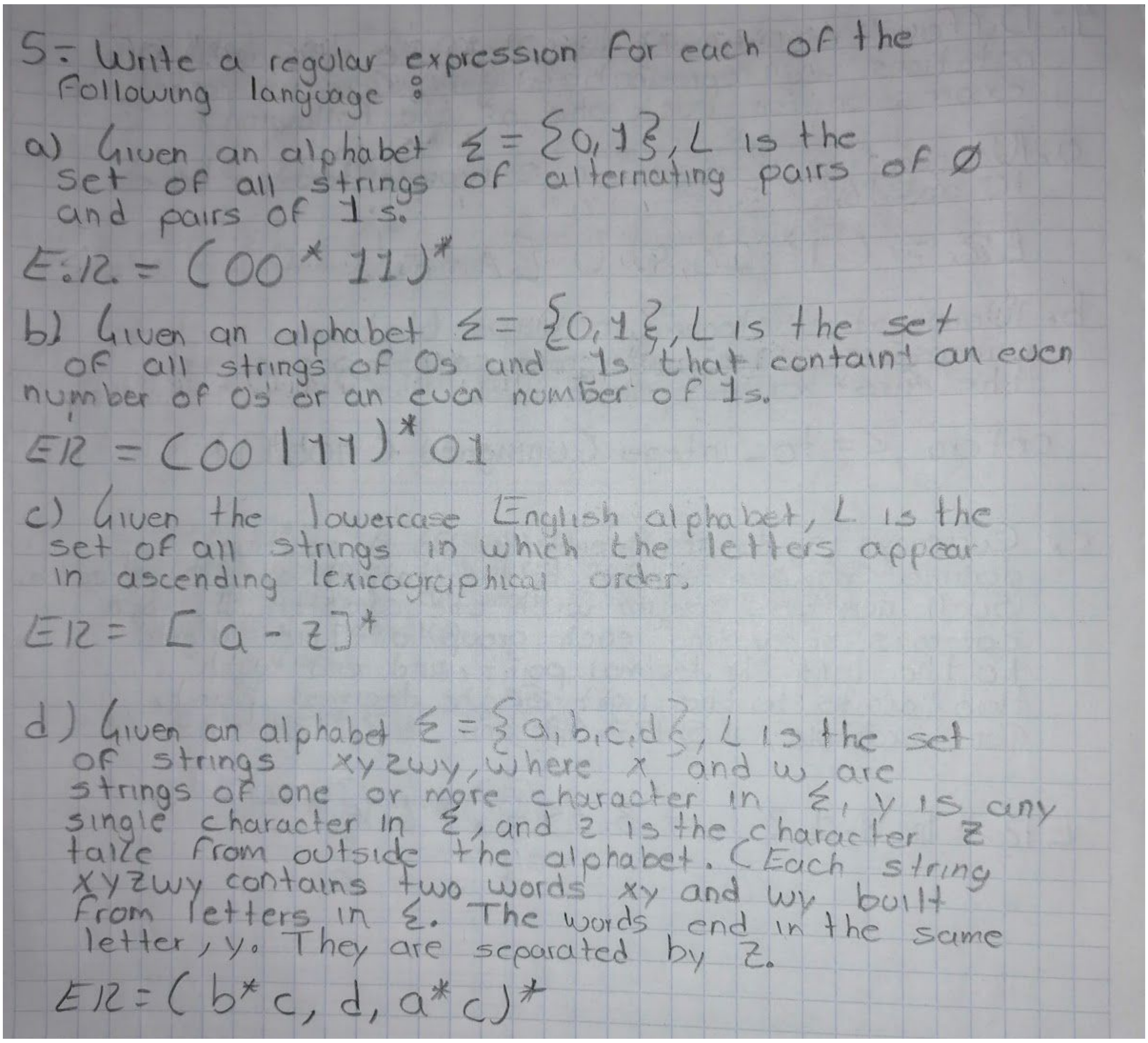
ER. = $(1^* [0,9) \cup [A-F].$

b. Nonnegative integers in VHDL that may include underscores (an underscore cannot occor as the first or last character).

entero <= to_integer (unsigned (1001))

c. Currency, in dollars, represented as a positive decimal number rounded to the nearest one-hundredth. Such numbers begin with the caracter $, have commas separating each group of three digits to the left the decimal point, and end with two digits to the right of the decimal point, For example, $ 8,937.13 and $ 7,777,777.77.

ER = $ 8,1,2 +, + 3.15 + . + 6,7



5= Write a regular expression for each of the following language:

a) Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of alternating pairs of Ø and pairs of 1s.

$E.R. = (00^{*}\ 11)^{*}$

b) Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of 0s and 1s that contain an even number of 0s or an even number of 1s.

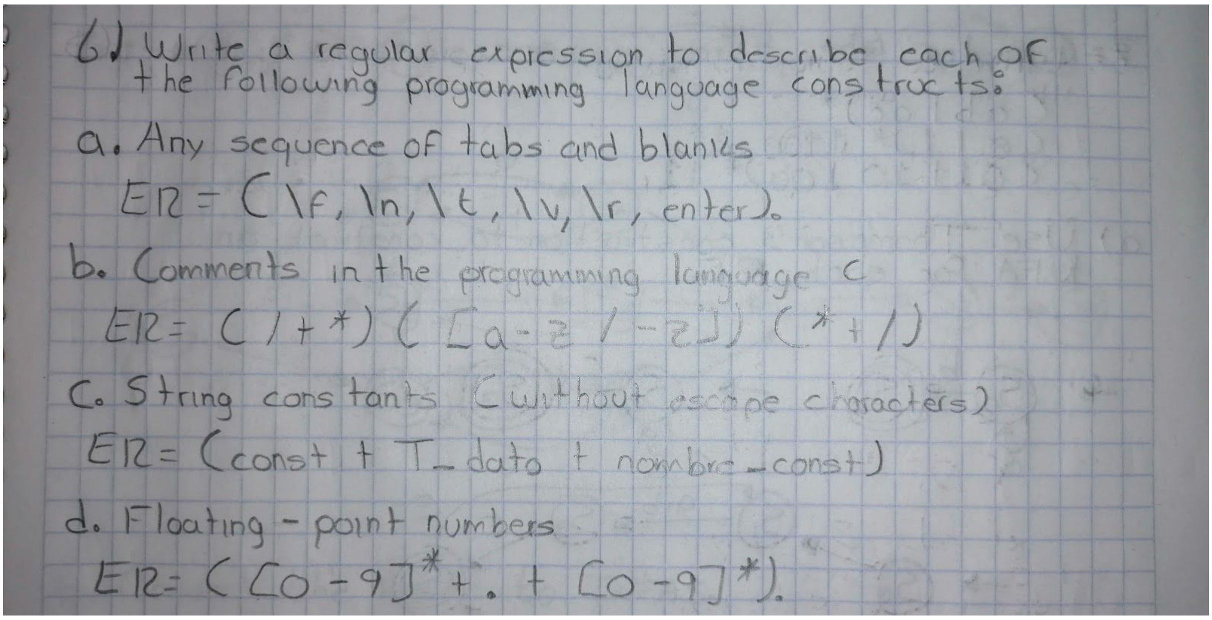
$ER = (00 \mid 11)^{*}\ 01$

c) Given the lowercase English alphabet, L is the set of all strings in which the letters appear in ascending lexicographical order.

$ER = [a - z]^{*}$

d) Given an alphabet $\Sigma = \{a, b, c, d\}$, L is the set of strings xyzwy, where x and w are strings of one or more character in Σ, y is any single character in Σ, and z is the character z take from outside the alphabet. (Each string xyzwy contains two words xy and wy built from letters in Σ. The words end in the same letter, y. They are separated by z.

$ER = (b^{*}c, d, a^{*}c)^{*}$



6) Write a regular expression to describe each of the following programming language constructs:

a. Any sequence of tabs and blanks

ER = (\f, \n, \t, \v, \r, enter).

b. Comments in the programming language C

ER = (/ + *) ([a - z / - z]) (* + /)

c. String constants (without escape characters)

ER = (const + T_dato + nombre_const)

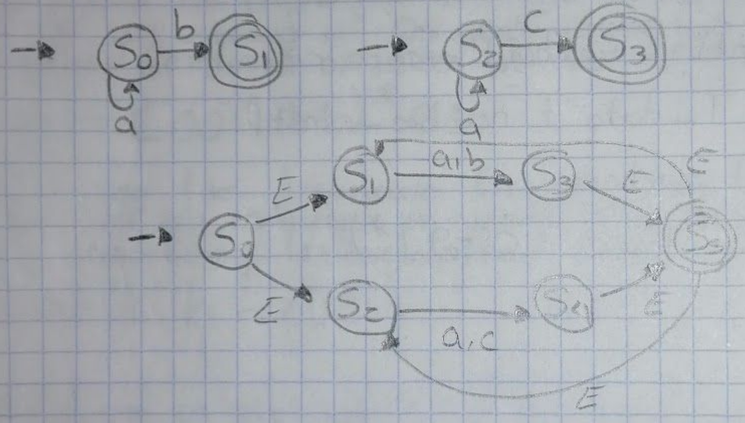
d. Floating-point numbers

$ER = ([0-9]^{*} + . + [0-9]^{*})$.

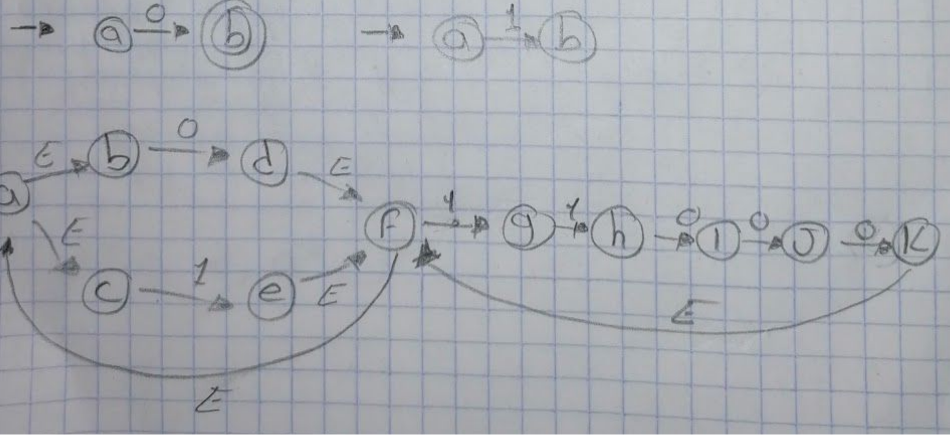
7: Consider the three regular expressions:

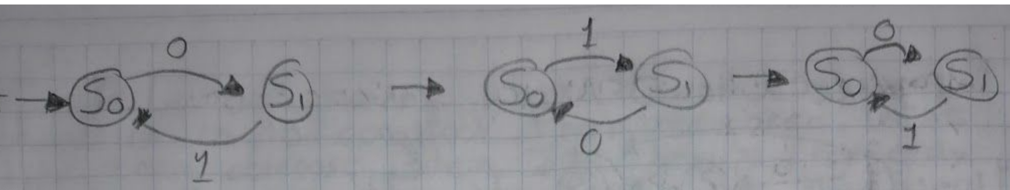
(ab | ac)*
(0 | 1)* 1100 1*
(01 | 10 | 00)* 11

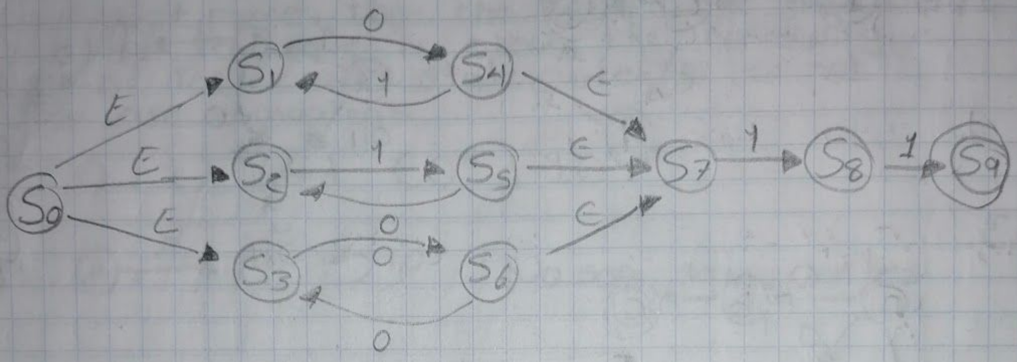
a) Use Thompson's construction to construct an NFA for each RE.



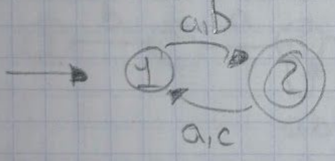
2.

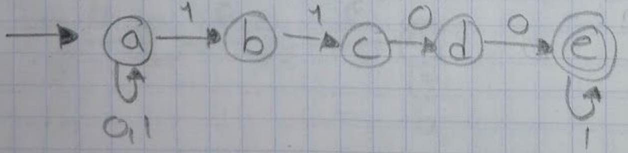


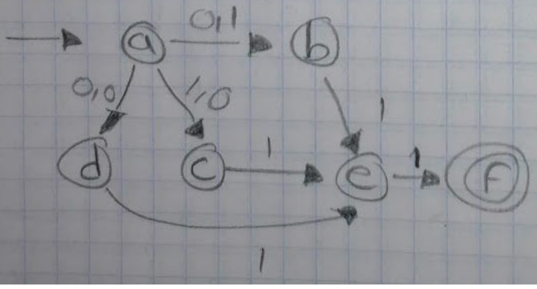
3÷ 

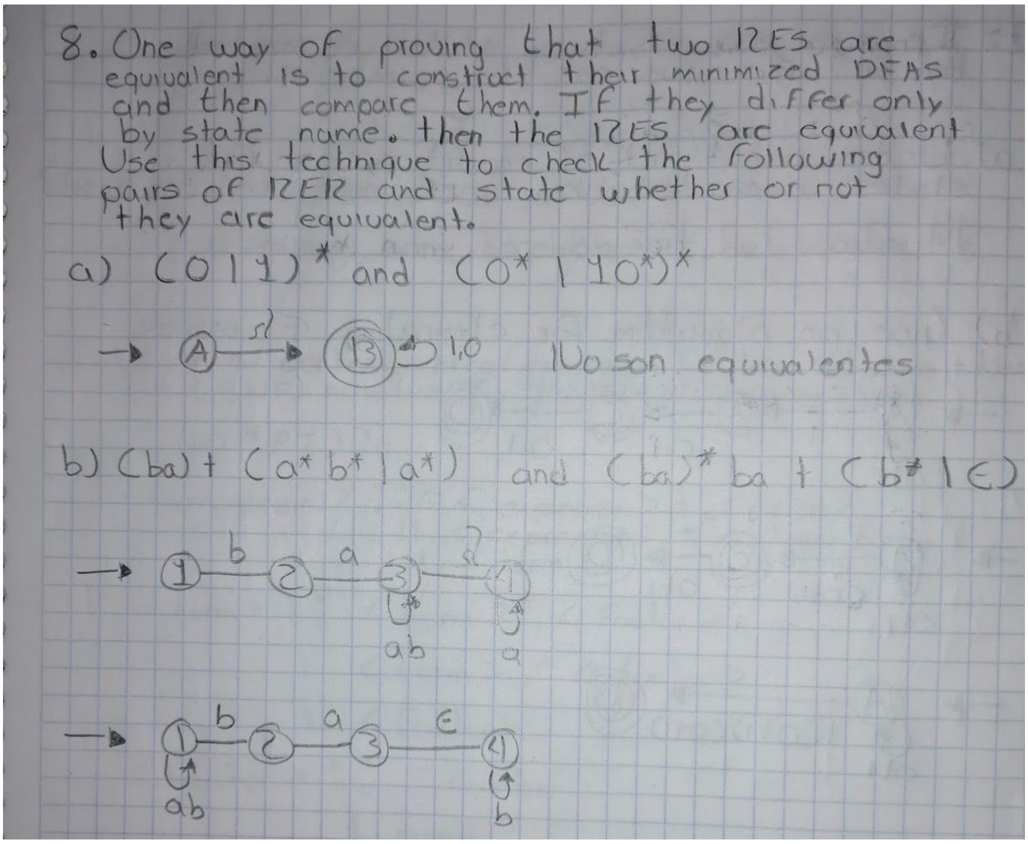


b. Convert the NFAS to DFAS.

1÷ 

2. 

3. 



8. One way of proving that two RES are equivalent is to construct their minimized DFAS and then compare them. If they differ only by state name. then the RES are equivalent Use this technique to check the following pairs of RER and state whether or not they are equivalent.

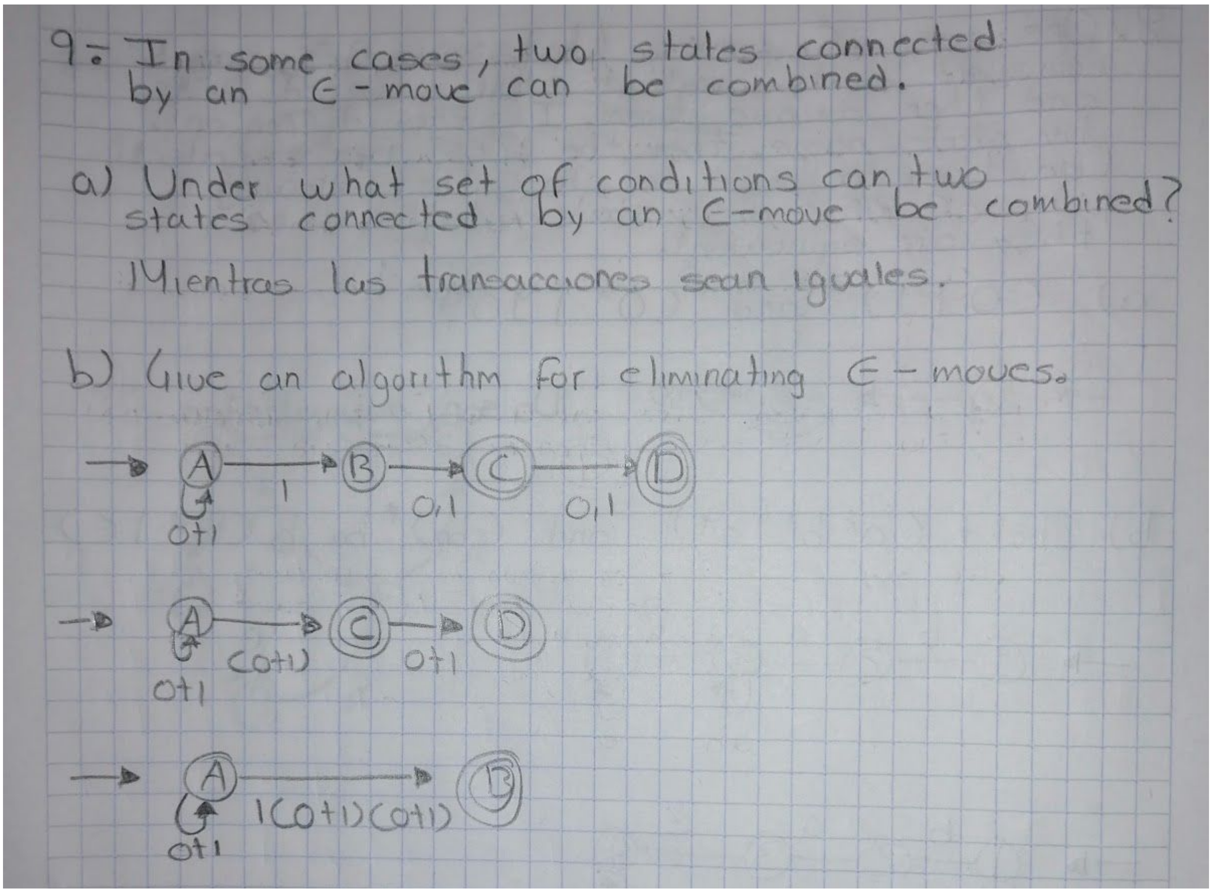
a) (0|1)* and (0* | 10*)*

A
B
1,0

No son equivalentes

b) (ba)+ (a* b* | a*) and (ba)* ba + (b* | ε)

1
b
2
a
3
ab
4
a
1
b
ab
2
a
3
ε
4
b



9.- In some cases, two states connected by an ε-move can be combined.

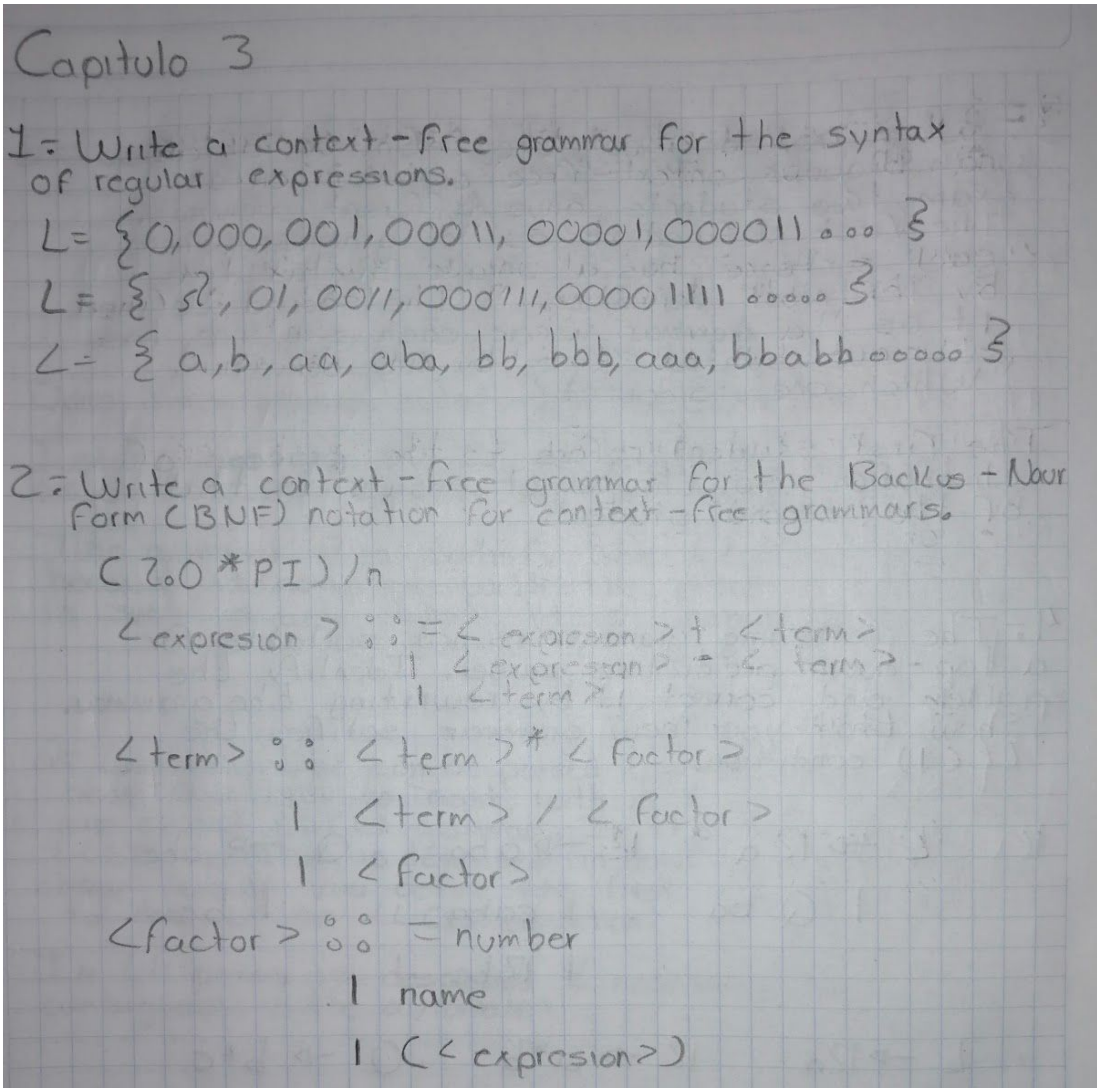
a) Under what set of conditions can two states connected by an ε-move be combined?

Mientras las transacciones sean iguales.

b) Give an algorithm for eliminating ε-moves.

A
0+1
1
B
0,1
C
0,1
D
A
0+1
(0+1)
C
0+1
D
A
0+1
1(0+1)(0+1)
B

CHAPTER 3



Capitulo 3

1: Write a context-free grammar for the syntax of regular expressions.

L= {0, 000, 001, 00011, 00001, 000011 ... }

L= { Ω, 01, 0011, 000111, 00001111 ..... }

L= { a, b, aa, aba, bb, bbb, aaa, bbabb ..... }

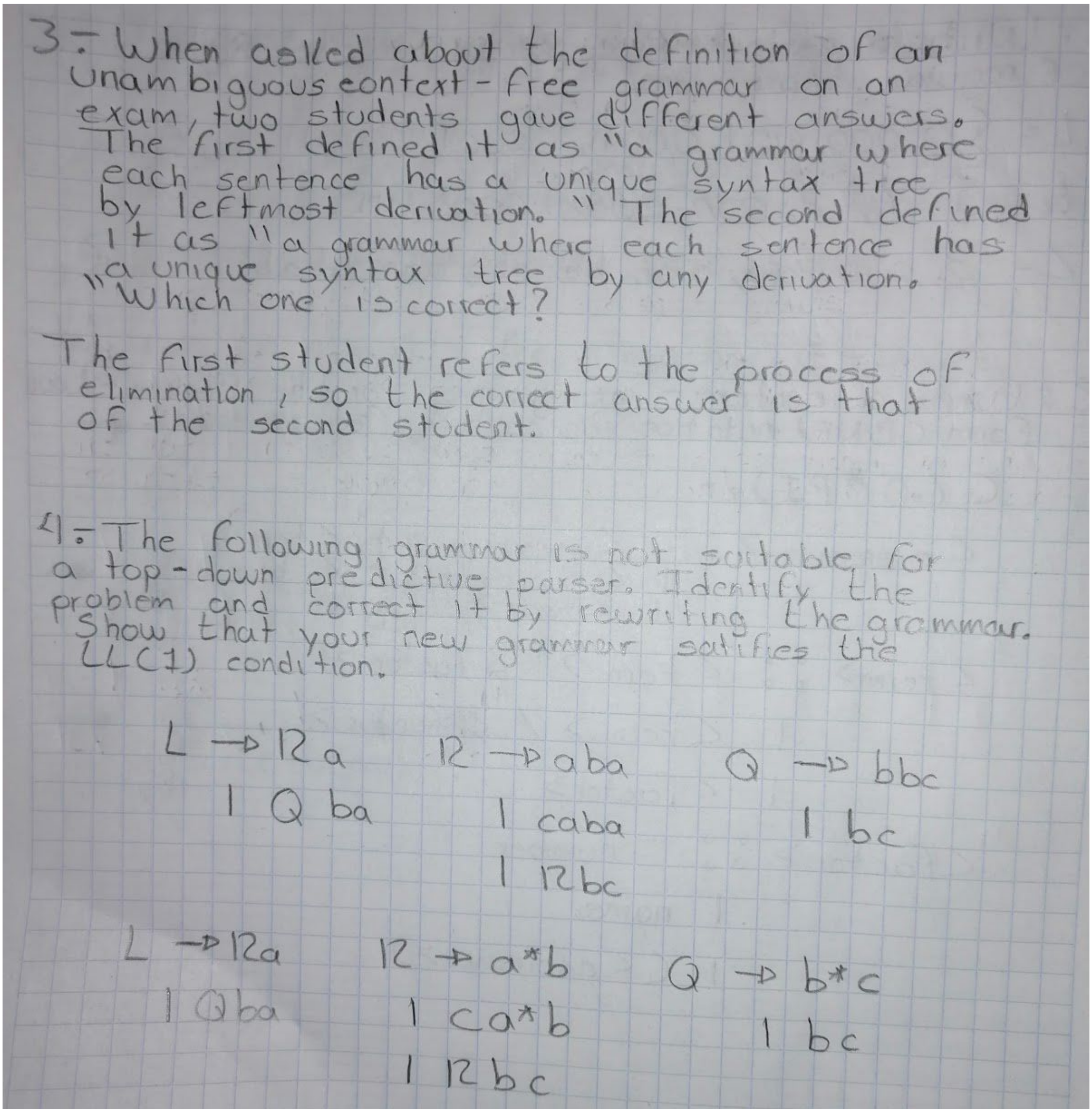
2: Write a context-free grammar for the Backus-Naur form (BNF) notation for context-free grammars.

```
(2.0 * PI) / n

<expresion> ::= <expresion> + <term>
              | <expresion> - <term>
              | <term>

<term> :: <term> * <factor>
        | <term> / <factor>
        | <factor>

<factor> :: = number
           | name
           | (<expresion>)
```



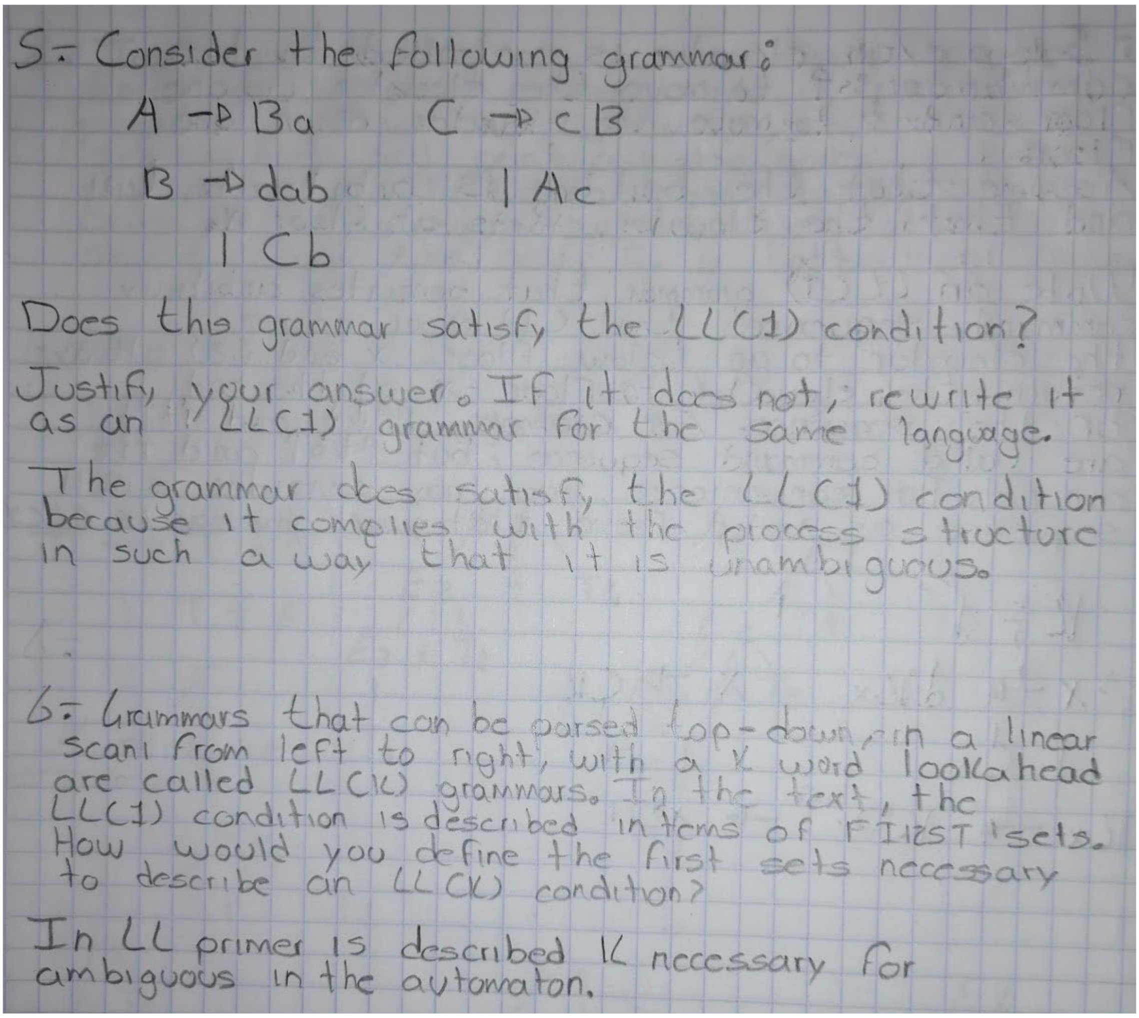
3.- When asked about the definition of an unambiguous context-free grammar on an exam, two students gave different answers. The first defined it as "a grammar where each sentence has a unique syntax tree by leftmost derivation." The second defined it as "a grammar where each sentence has a unique syntax tree by any derivation." Which one is correct?

The first student refers to the process of elimination, so the correct answer is that of the second student.

4.- The following grammar is not suitable for a top-down predictive parser. Identify the problem and correct it by rewriting the grammar. Show that your new grammar satifies the LL(1) condition.

L → Ra   R → aba   Q → bbc
  | Qba | caba | bc
              | Rbc

L → Ra   R → a*b   Q → b*c
  | Qba | ca*b | bc
              | Rbc



5.- Consider the following grammar:

A → Ba   C → cB
B → dab | Ac
  | Cb

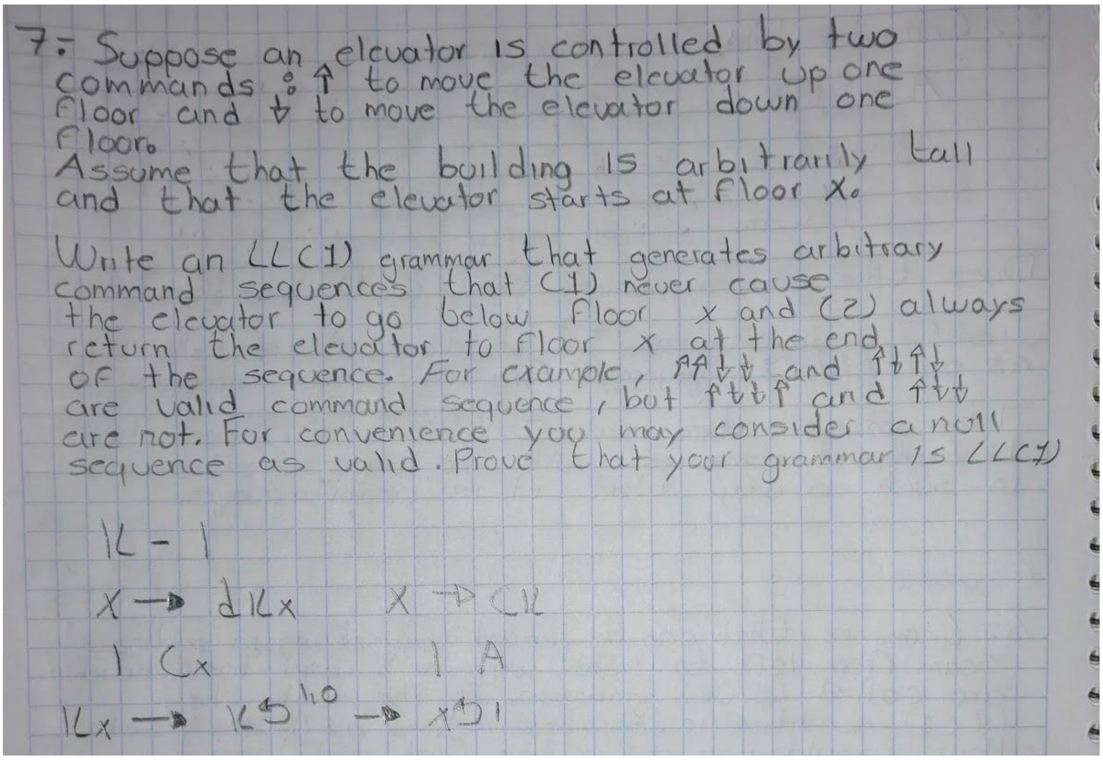
Does this grammar satisfy the LL(1) condition?

Justify your answer. If it does not, rewrite it as an LL(1) grammar for the same language.

The grammar does satisfy the LL(1) condition because it complies with the process structure in such a way that it is unambiguous.

6.- Grammars that can be parsed top-down, in a linear scan from left to right, with a k word lookahead are called LL(K) grammars. In the text, the LL(1) condition is described in terms of FIRST sets. How would you define the first sets necessary to describe an LL(K) condition?

In LL primer is described K necessary for ambiguous in the automaton.



7.- Suppose an elevator is controlled by two commands: ↑ to move the elevator up one floor and ↓ to move the elevator down one floor.
Assume that the building is arbitrarily tall and that the elevator starts at floor X.

Write an LL(1) grammar that generates arbitrary command sequences that (1) never cause the elevator to go below floor x and (2) always return the elevator to floor x at the end of the sequence. For example, ↑↑↓↓ and ↑↓↑↓ are valid command sequence, but ↑↓↓↑ and ↑↓↓ are not. For convenience you may consider a null sequence as valid. Prove that your grammar is LL(1)

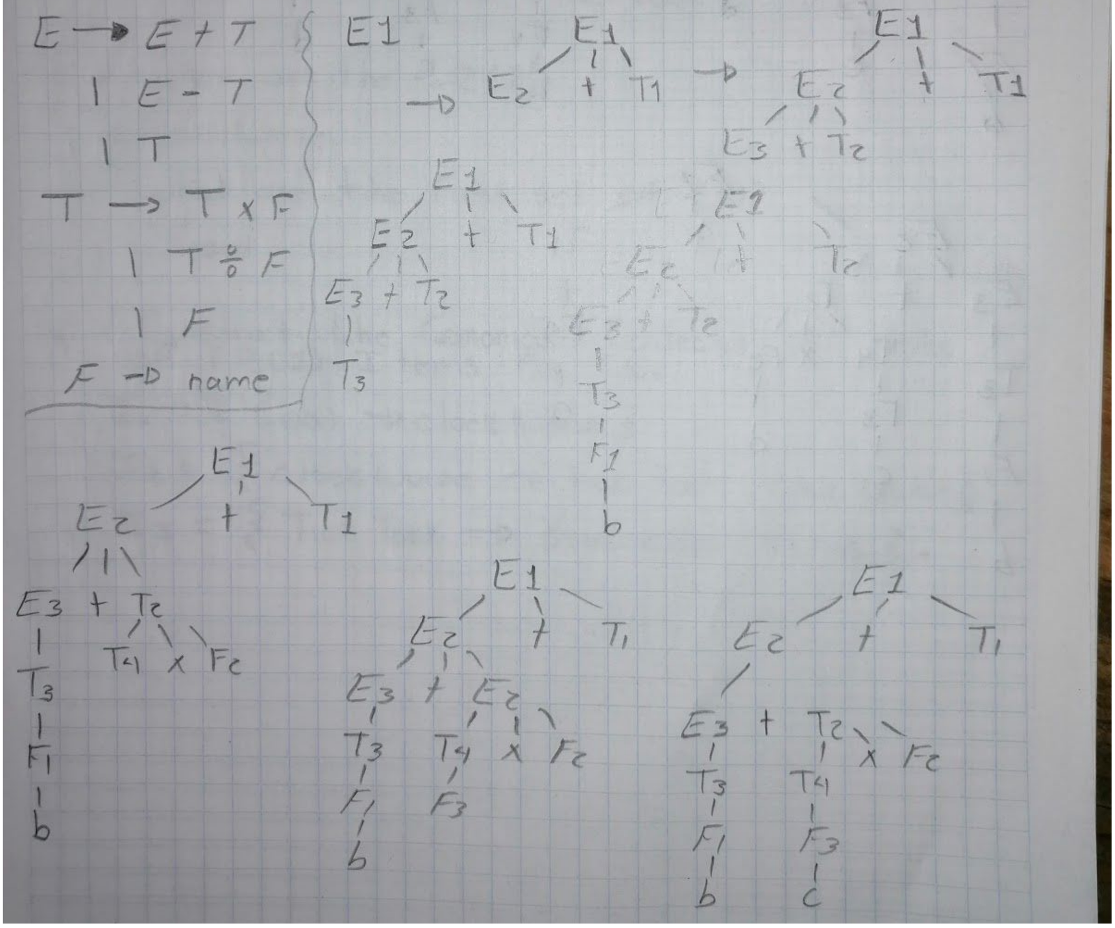
K - 1

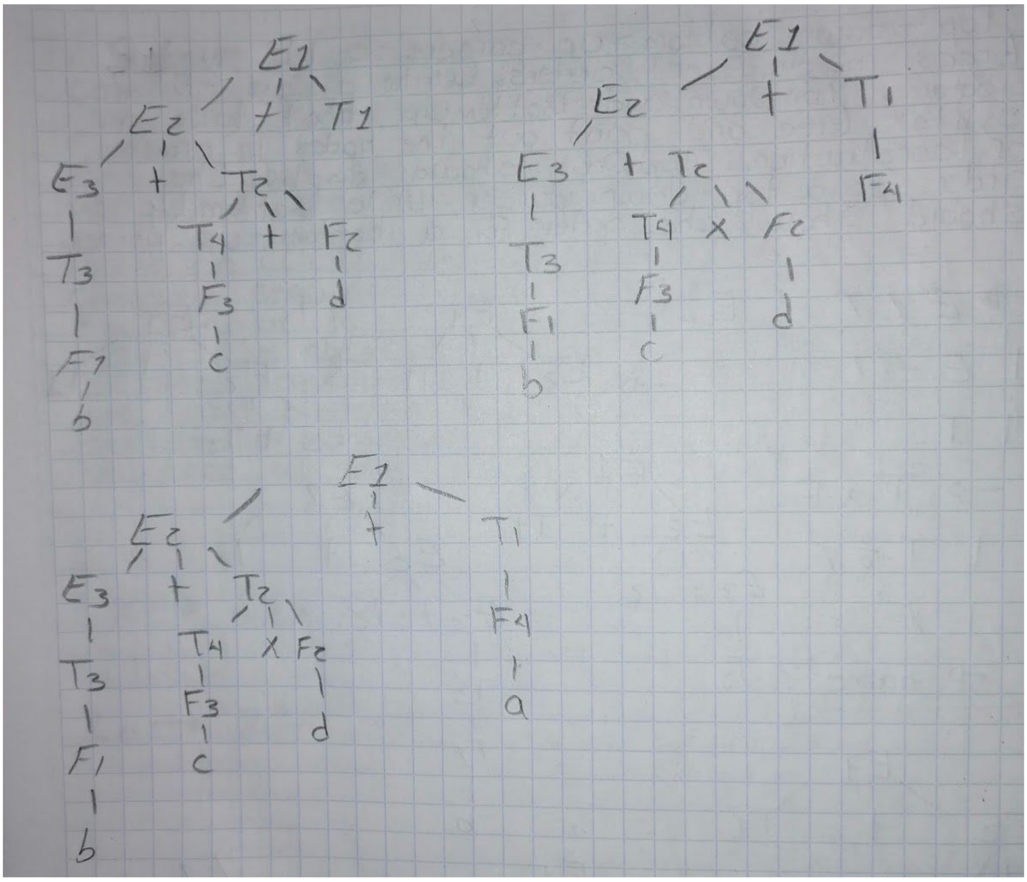
X → dKx   X → CK
  | Cx | A

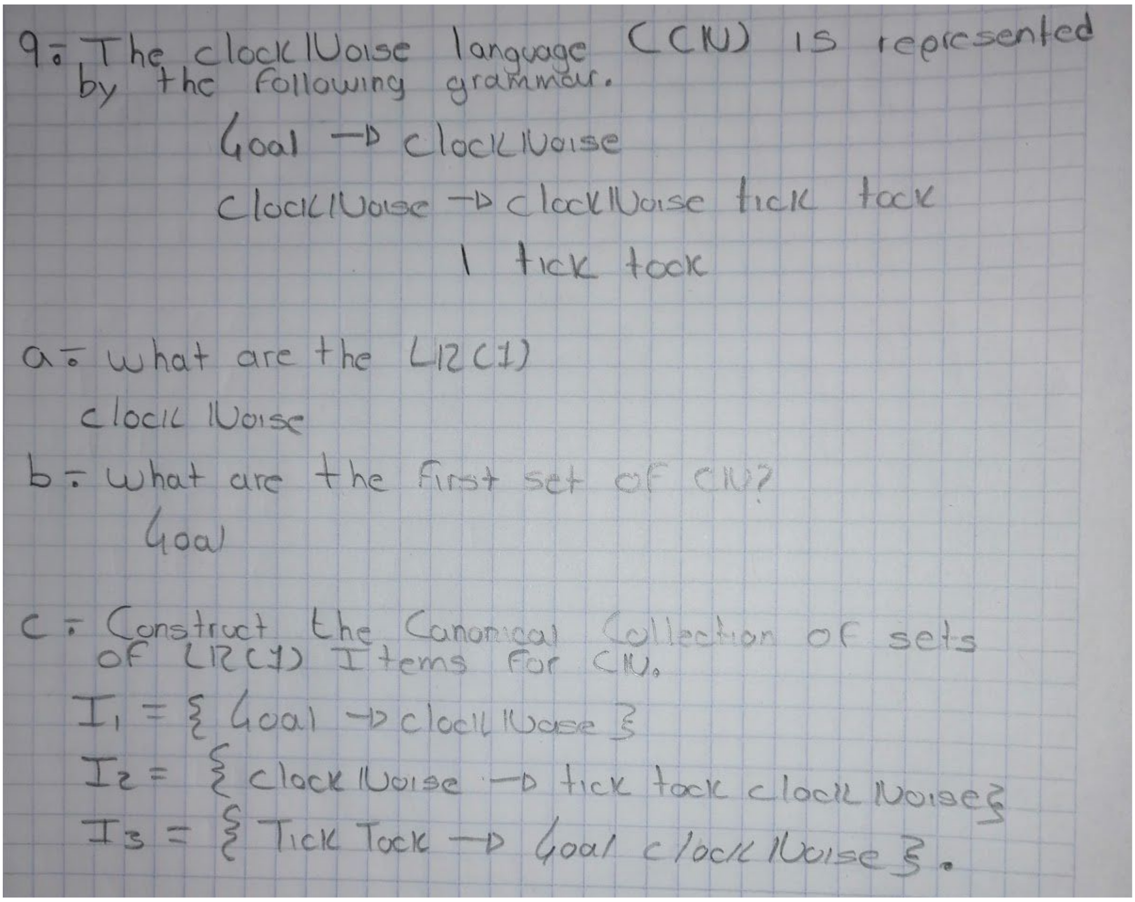
Kx → K5 1,0 → X5 1

8: Top-down bottom-up parsers build syntax trees in different orders. Write a pair of programs, TopDown and BottomUp, that take a syntax tree and print out the nodes in order of construction. TopDown should display the order for a top-down parser, while BottomUp should show the order for a bottom-up parser.

E → E + T
  | E - T
  | T
T → T x F
  | T ÷ F
  | F
F → name







9. The clockNoise language (CN) is represented by the following grammar.

Goal → clockNoise

ClockNoise → clockNoise tick tock

| tick tock

a. what are the LR(1)

clock Noise

b. what are the First set of CN?

Goal

c. Construct the Canonical Collection of sets of LR(1) Items for CN.

I_1 = { Goal → clockNoise }

I_2 = { clockNoise → tick tock clockNoise }

I_3 = { Tick Tock → Goal clockNoise }.

ACTIVITIES WEEK 13 (DIC 14-18, 2020)

AN INTRODUCTION TO TYPE SYSTEMS

Most programming languages associate a collection of properties with each data value. This collection of properties is called the type of the value.

Types can be specified by membership; for example, an integer could be any integer i in the range -2, 31 ≤ i < 2 31, or red could be a value in a type of enumerated colors, defined as the set

{red, orange, yellow, yellow, green, blue, blue, brown, black, white}.

Types may be specified by rules; for example, the declaration of a structure in c defines a type.

The set of types in a programming language, together with the rules that use types to specify program behavior, are collectively called a type system.

ACTIVITIES WEEK 14 (ENE 7-8, 2021)

INTRODUCTION TO INTERMEDIATE REPRESENTATIONS

Compilers are usually organized as a series of passes. As the compiler obtains knowledge about the code it compiles, it must pass that information from one step to the next. Thus, the compiler needs a representation for all the facts that are derived from the program.

We call this representation an intermediate representation, or ir. A compiler may have a single ir, or it may have a series of irs that it uses when transforming the source code into its target language.

The compiler does not refer back to the source text; instead, it looks at the ir form of the code. The properties of a compiler's irs have a direct effect on what the compiler can do to the code.

A compiler needs a representation for all knowledge that is derived about the program being compiled Most passes in the compiler consume ir; the scanner is an exception. Most passes in the compiler produce go; passes in the code generator can be exceptions.

ACTIVITIES WEEK 15 (ENE 11-15, 2021).

INTRODUCTION TO THE PROCEDURE ABSTRACTION.

The procedure is one of the central abstractions in most modern programming languages. Procedures create a controlled execution environment; each procedure has its own private storage.

Procedures help define interfaces between system components; interactions between components are typically structured through procedure calls.

Conceptual roadmap.

To translate a program from source language to executable code, the compiler must map all the source language constructs that the program uses into operations and data structures in the target processor.

The compiler needs a strategy for each of the abstractions supported by the source language. The strategies include both algorithms and data structures that are embedded in the executable code.