Iztapalapa Technological Institute

Computer Systems engineer

Languages and automata 2

Santana González Jesús Salvador: 171080127

Cabrera Ramírez Gerardo: 171080187

Morales Carrillo Gerardo: 171080120

Week 14 activity

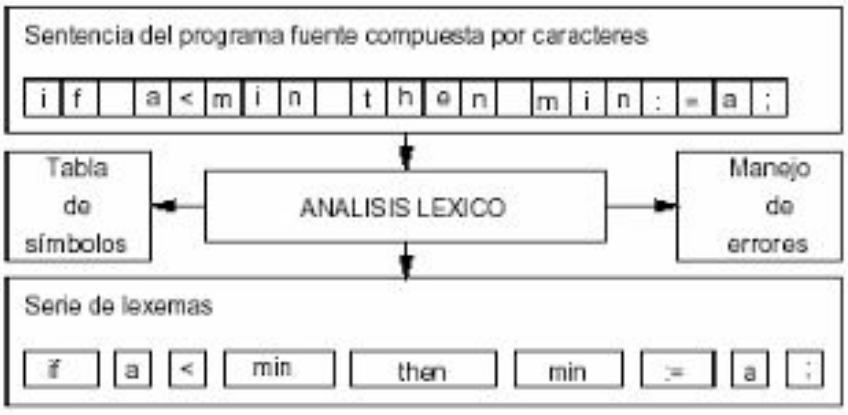
10/16/2020

"Lexical Analysis Part 1"

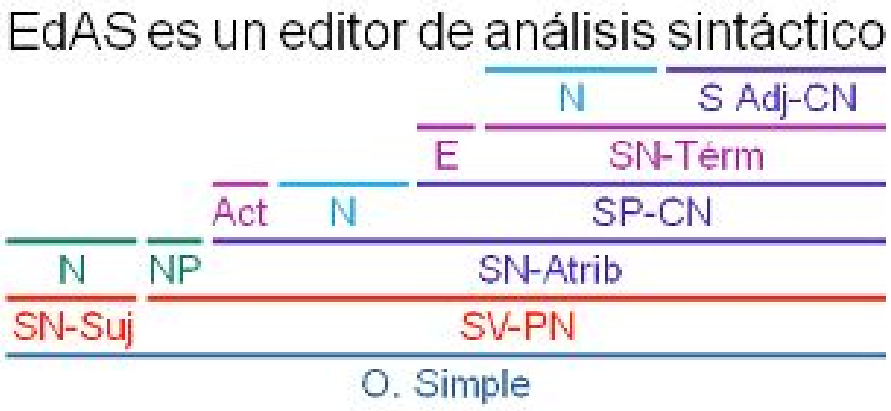
Concepts:

Lexical Analysis

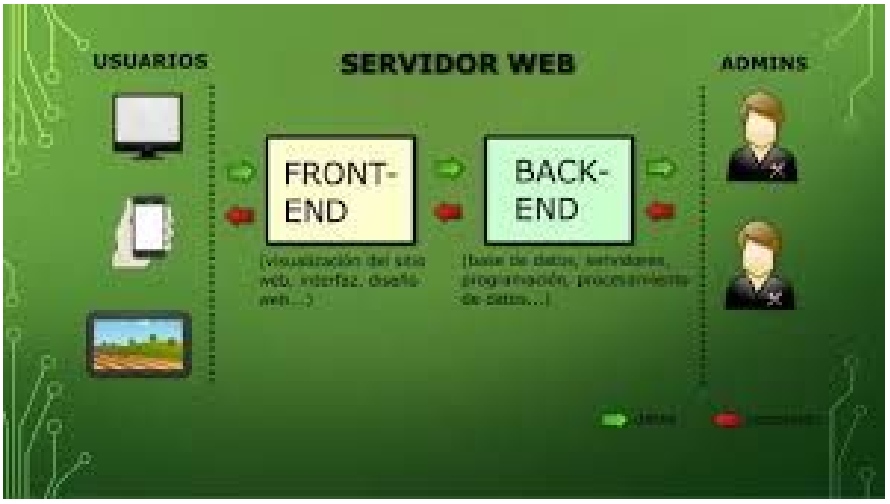
The first phase of a compiler is called Lexical analysis or scanning. The Lexical Analyzer reads the flow of characters that make up the source program and groups them into sequences.



parsing the second phase of the compiler is the syntactic or parsing analysis. the parser (parser) uses the first components of tokens produced by the lexical analyzer to create an intermediate representation in a tree that describes the grammatical structure of the token flow.

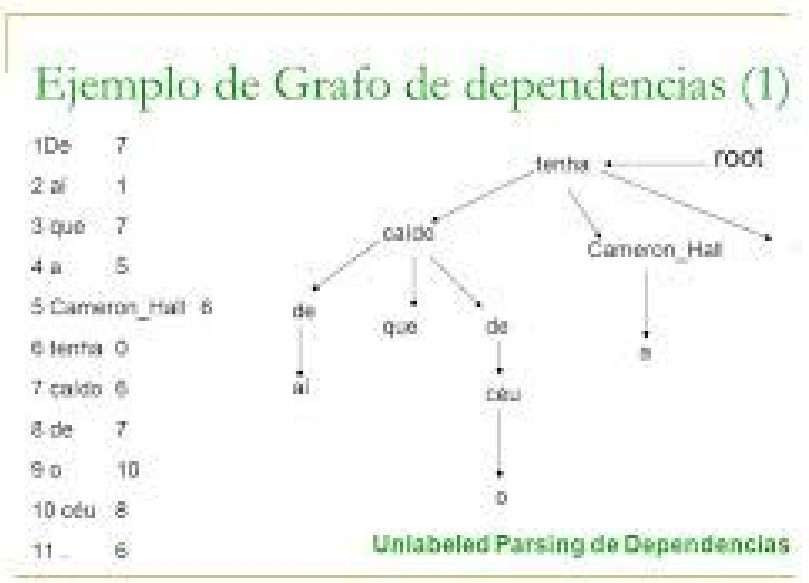


Front End**:** The Front End is the part of the compiler that interacts with the user and is generally independent of the platform on which the user works.



Back End: This part of the compiler is responsible for generating the code in machine format, from the work done by the Front End

Dependency graphs A dependency graph describes the information flow between the attribute instances in a specific parse tree; an arrow from one attribute instance to another means that the value of the first is needed to calculate the second.



management Error information.

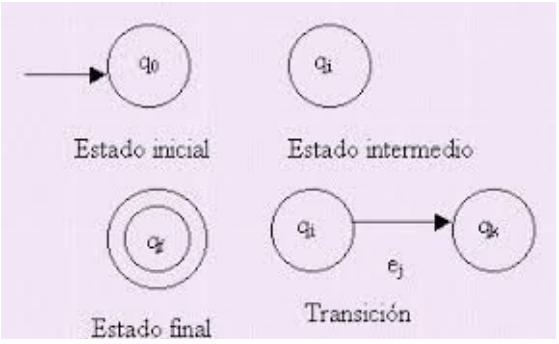
If compilers had to process only correct programs, their design and implementation will be greatly simplified. But programmers write bad programs frequently, and a good compiler should help the programmer to locate and identify errors.

Generation of the object code.

The final phase of a compiler is the generation of the object code, consisting of machine code or assembly code.

Language representation:

In general, there are two different schemes to define a Language, which are known as the generator scheme and the recognizing scheme. In the case of generator schemes, it is a mechanism that allows us to "generate" the different sentences of the language.



Concept of grammar:

Grammar is a mathematical entity or model that allows specifying a language, that is, it is the set of rules capable to generate all the combinatorial possibilities of that language, and only those of that language, be it a formal language or a natural language.

Santana Gonzales Jesús Salvador

"Lexical Analysis (parts 2 and 3)" and "Syntactic Analysis (part 1)"

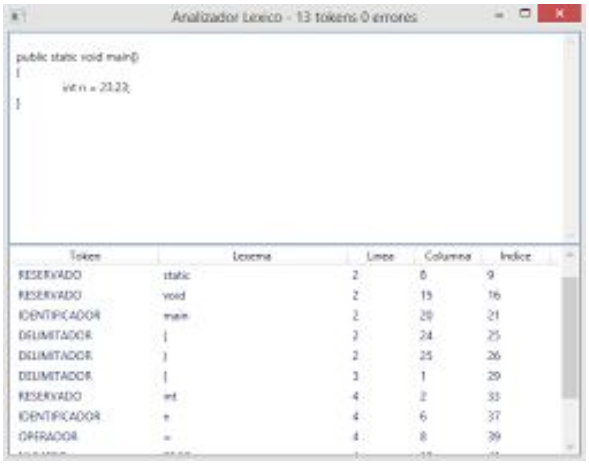
Functions of the lexical analyzer and their advantages
The lexical analyzer performs several functions, the fundamental one being that of grouping the characters that it is reading one by one of the source program and form the tokens.

Implementation of a lexical analyzer There are several ways to implement a lexical analyzer:

Using a generator of lexical analyzers: these are tools that from regular expressions generate a program that allows to recognize the tokens or lexical components. These programs are usually written in C, where one of the tools is FLEX, or they can be written in Java, where the tools can be JFLEX or JLEX.

Using a high-level language: the lexical analyzer is programmed from the transition diagram and the corresponding pseudo-code (see an example in Louden, 2004).

Lexical
errors Lexical errors are detected when, during the character recognition process, the symbols that we have in the input do not match any pattern.



What is the parser?

It is the analyzer phase that is in charge of checking the input text based on a given grammar. And in case the input program is valid, it supplies the syntactic tree that recognizes it. In theory, the output of the parser is assumed to be some representation of the parser that recognizes the sequence of tokens supplied by the parser. In practice, the parser also does:

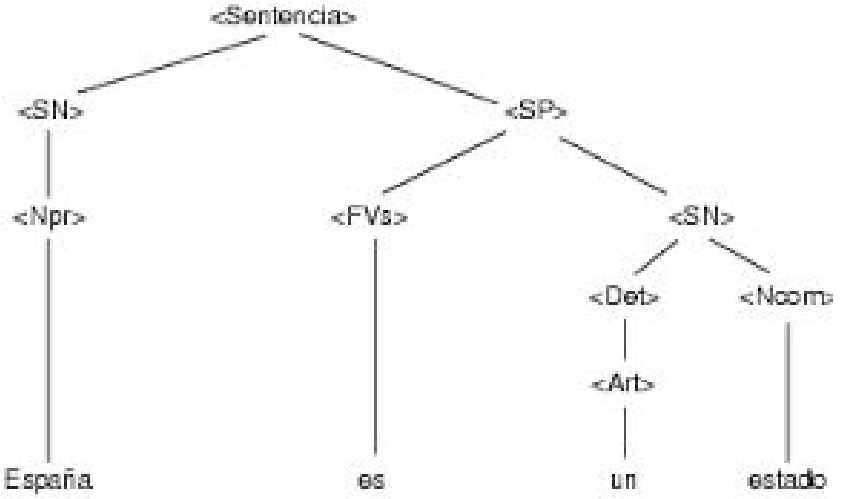
• Access the symbol table (to do part of the work of the semantic parser).
• Type checking (from the semantic analyzer).
• Generate intermediate code.
• Generate errors when they occur.

In short, it performs almost all the operations of the compilation. This working method leads to syntax-driven compilation methods.

Syntactic error handling

If a compiler had to process only correct programs, its design and implementation would be greatly simplified. But programmers often write bad programs, and a good compiler should help the programmer identify and locate errors. What's more, considering error handling early on can simplify the structure of a compiler and improve its response to errors. Programming errors can be of the following types:

• Lexical, produced by misspelling an identifier, a keyword or an operator.
• Syntactic, by an arithmetic expression or unbalanced parentheses.
• Semantics, as an operator applied to an incompatible operand.
• Logical, it can be an infinitely recursive call.



Santana Gonzalez Jesus Salvador

Part 4

SEMANTIC ANALYSIS IN LANGUAGE PROCESSORS

The semantic analysis phase of a language processor is that which computes the additional information necessary for the processing of a language, once the syntactic structure of a program has been obtained. It is therefore the phase after the syntactic analysis and the last in the synthesis process of a programming language.
Syntax of a programming language is the set of formal rules that specify the structure of the programs belonging to that language. Semantics of a programming language is the set of rules that specify the meaning of any syntactically valid statement. Finally, the semantic analysis1 of a language processor is the phase in charge of detecting the semantic validity of the sentences accepted by the parser.

The syntax of the C language indicates that expressions can be formed with a set of operators and a set of basic elements. Operators, with infix binary syntax, include assignment, product, and division. Among the basic elements of an expression are identifiers and unsigned integer constants (among others).
Its semantics identify that in the record associated with the surface identifier the resulting value of the product of the values associated with base and height, divided by two (the surface of a triangle) will be associated.

Semantic Specification of Programming Languages

There are two ways to describe the semantics of a programming language: through informal or natural and formal specification. The informal description of a programming language is carried out through natural language. This makes the specification intelligible (in principle) to anyone.

Experience tells us that it is a very complex, if not impossible task to describe all the characteristics of a programming language in a precise way. As a particular case, see the ISO / ANSI C ++ language specification [ANSIC ++]. The formal description of the semantics of programming languages is the rigorous description of the meaning or behavior of programs, programming languages, abstract machines or even any hardware device.

- Reveal possible ambiguities in existing language processor implementations or in descriptive documents of programming languages.
- Be used as a basis for the implementation of language processors.
- Verify program properties in relation to correction tests or information related to their execution.
- Design new programming languages, allowing the recording of decisions on particular language constructions, as well as allowing the discovery of possible irregularities or omissions.

- Facilitate understanding of languages by the programmer and as a communication mechanism between language designer, implementer and programmer. The semantic specification of a language, as a reference document, clarifies the behavior of the language and its various constructions.
- Standardize languages by publishing their semantics in an unambiguous way. Programs must be able to be processed in another processor implementation of the same language exhibiting the same behavior.

Formal Specification of Semantics

Although the formal specification of the syntax of a language is usually carried out through the standard description of its grammar in BNF (Backus-Naur Form) notation, in the case of semantic specification the situation is not so clear ; there is no globally extended standard method. The behavior of the different constructions of a programming language can be described from different points of view.

Tasks and Objectives of Semantic Analysis

The semantic analysis11 of a language processor is the phase in charge of detecting the semantic validity of the sentences accepted by the parser.

Checks postponed by the parser

When implementing a processor for a programming language, it is common to find situations in which a context-free grammar can syntactically represent properties of the language; however, the resulting grammar is complex and difficult to process in the parsing phase. In these cases it is common to see how the compiler developer writes a simpler grammar that does not represent details of the language, accepting them as valid when they do not really belong to the language.

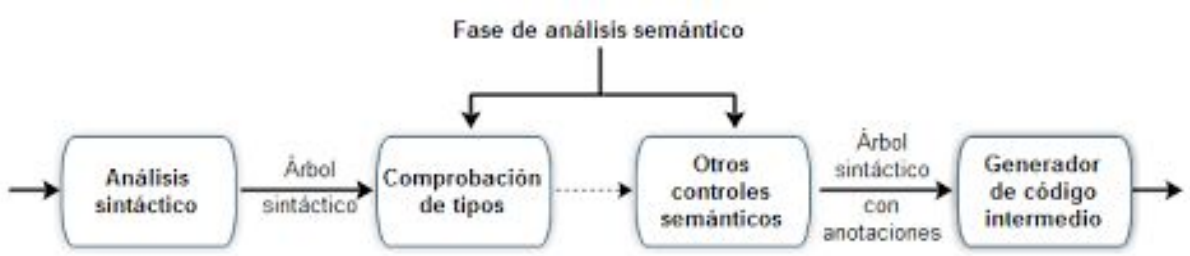
Dynamic

checks All the semantic checks described in this point are usually carried out in the compilation phase and are therefore called "static". There are checks that, in their most general case, can only be carried out at runtime and therefore are called "dynamic". These are usually checked by an interpreter or by compiler-generated checking code - it may also be the case that they are not checked. Several examples can be access to an out of range vector, use of a null pointer or division by zero.

such

Surelychecks,such checks is the most comprehensive and extensive semantic analysis phase. Whether in a static (compile-time), dynamic (run-time) mode, or both, type checks are required in every high-level language. Briefly, the semantic analyzer should carry out the following two type-related tasks: Tasks and Objectives of Semantic Analysis.

Check the operations that can be applied to each construction of the language. Given an element of language, its type identifies the operations that can be applied to it. For example, in the Java language the product operator is not applicable to a reference to an object. Otherwise, the dot operator is valid. 2. Infer the type of each language construction. In order to implement the above check, it is necessary to know the type of every syntactically valid construction of the language. Thus, the semantic analyzer must apply the different type inference rules described in the programming language specification, in order to know the type of each language construction.



Santana Gonzalez Jesus Salvador

PART 5

Syntactic Analysis.
Its function is to label each of the syntactic components that appear in the sentence and analyze how the words are combined to form grammatically correct constructions. The result of this process consists of generating the structure corresponding to the syntactic categories formed by each of the lexical units that appear in the sentence. Grammars, as shown in the following figure, are formed by a set of rules which are:

43 O -> SN,
SV SN -> Det,
N SN -> Proper Name
SV -> V,
SN SV -> V SP -> Preposition,
SN SN = noun phrase

SV = verb phrase Det = determinant

The lexical analyzer has as input the source code in the form of a sequence of characters, the parser has as input the lexemes that it provides the lexical analyzer and its function is to check that they are ordered correctly (depending on the language we want to process) the two analyzers usually work together and even the lexicon is usually a subroutine of the syntactic

Semantic Analysis.
It is the extension of the syntactic analysis for the understanding of semantic networks and the knowledge base both to see if it makes sense and that it is directed to natural language since that is what it focuses on, to put it in other words is to give it meaning associated with the formal structures of language.
This analyzer computes additional information necessary for the processing of a language once it is taking a complete and exact analysis of the parser well and generates true results. Its primary or main objective of the semantic analyzer is that the analyzed program satisfies the rules required by the language specification to ensure its correct execution.

Advantages of semantic analysis

- The formal description of the semantics of programming languages is the rigorous description of the meaning or behavior of programs, the programming language in addition to verifying properties of relation with proofs of correctness or information related to its execution.
- Be used as a basis for the implementation of language processors.
- To facilitate the understanding of languages by the programmer and as a communication mechanism between the language designer and implementer, the semantic specification of a language as a reference document clarifies the behavior of the language and its various constructions.
- It reveals possible existing ambiguities.
- Create language processor implementations or descriptive documents of natural language processing.

The semantic analyzer detects the semantic validity of the statements accepted by the parser. The semantic analyzer usually works simultaneously with the parser and in close cooperation semantics is understood as the set of rules that specify the meaning of any syntactically correct sentence written in a given language.

The semantic routines must perform the evaluation of the grammar attributes following the semantic rules associated with each production of the grammar.

Syntactic analysis is the phase in which it is tried to determine the type of the intermediate results, to verify that the arguments that an operator has belong to the set of possible operators and if they are compatible with each other, etc. In short, it will verify that the meaning of what is being read is valid.

The “theoretical” output from the semantic analysis phase would be a semantic tree. It consists of a syntactic tree in which each of its branches has acquired the meaning it should have. In the case of polymorphism operators (a single symbol with several meanings), the semantic analysis determines which one is applicable.

It is made up of a set of independent routines, called by morphological and parsers. Semantic analysis uses as input the syntactic tree detected by the syntactic analysis to check type constraints and other semantic constraints and ready the code generation. Semantic routines usually make use of a stack (the semantic stack) that contains the semantic information associated with the operands in the form of semantic registers.

Front end

The Frontend makes use of technologies or languages of style or programming on the client side for the structuring, layout and animation of websites. The style or programming languages to which we refer are: HTML (hypertext markup language), CSS (cascading style sheets) and JavaScript, among others:

HTML is a markup language that serves to define the structure of the content of your website.
CSS is a style language that is used to encode the structure created by HTML (give text color, include margins, change the typography of the content ...).
JavaScript is a programming language with which you can program the interaction with the user.

Once these three terms are clarified, what is the Frontend? The Frontend is a technology that is responsible for the design of a web page, that is, it takes care of the aesthetics, drafts, mockups (scale or real-size design mockup), interface, user usability, structure, colors, fonts and effects. Also, through JavaScript is the possibility of scheduling events, and validating contact forms, among other functions.

In other words, the Frontend is the part of the software in charge of interacting with the visitors of the web.

It is the component part visible to the user. The front-end in software design and web development refers to the visualization of the navigating user or, in other words, it is the part that interacts with users.

The Frontend covers those technologies that refer exclusively to the visual comfort of the user. Stylize the website using the techniques offered by the User Experience (user experience). You will also learn about web design so that the structure of the site facilitates an orderly visualization and comfort for whoever is browsing the page. In addition, it will take care of generating an intuitive website for the user.

These technologies are generalized in the following programming languages:

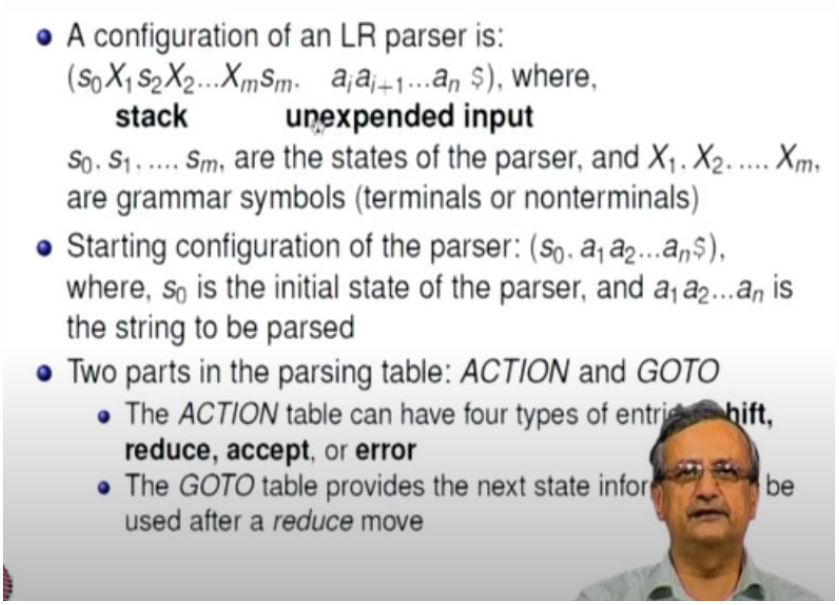
- HTML: In charge of ordering the content of a website.
- CSS: It is part of graphic design and deals with the creation and structures of web documents.
- JAVA SCRIPT.:Allows the creation of complex activities in web development such as instant updates, maps, infographics, 3D, etc.

The programmer in charge of the frontend will handle only these 3 technologies, also knowing perfectly about the backend even if he does not work with it, since they are closely related and one needs the other.

In short, this web technology will visually optimize the page, structure it comfortably in the eyes of the user and maintain an adequate aesthetic of the site so that the interaction on it (consultation, request, clicks) is correct and efficient. Therefore, one of the most valuable qualities to obtain a responsive website is creativity to be able to design eye-catching and attractive sites that are optimal for all types of devices and resolutions.

The components found on the front of the system are as follows:

- usability and accessibility tests;
- design and markup languages such as HTML, CSS, and JavaScript;
- graphic design and image editing tools;
- search engineor optimizationSEO;
- web performance and browser compatibility.



Santana Gonzalez Jesus Salvador

Part 6

Lexical, Syntactic and Semantic Analyzer

Computers are a balanced mix of Software and Hardware

Compilers are computer programs that translate a language written in source language from one language to another and produce an equivalent program written in object language

A compiler is internally composed of several stages or phases that perform logical operations and these are:

Lexical analyzer

Reads the sequence of characters from left to right of the source program and groups the sequence of characters into units with their own meaning (lexical components or tokens)
. keywords identifiers, operators, numeric constants, punctuation marks such as sentence separators, curly braces, paresthesias, and so on. are various classifications of lexical components.

Lexical analysis (Scanner)
Scanner has the functions of reading the source program as a character file and dividing it into tokens. Tokens are the reserved words of a language, a sequence of characters that represents a unit of information in the source program. In each case a token represents a certain pattern of characters that the lexical analyzer recognizes, or adjusts from the beginning of the input characters. In this way, it is necessary to generate a computational mechanism that allows us to identify the transition pattern between the input characters, generating tokens, which will later be classified. This mechanism can be created from a specific type of state machine called a finite automaton.

Representation of a Lexical Analyzer Lexical
components are represented:
1. Reserved words: if, while, do,…
2. Identifiers: variables, functions, user-defined types, labels,…
3. Operators: =,>, <, > =, <=, +, *,…
4. Special symbols:;, (), {},…
5. Numeric constants. literals that represent integer and float values.

6. Character constants: literals that represent strings of characters.

What is a lexical analyzer?
It is in charge of searching for the lexical components or words that make up the source program, according to rules or patterns. The input of the lexical analyzer can be defined as a sequence of characters.
The lexical analyzer has to divide the sequence of characters into words with their own meaning and then convert it to a sequence of terminals from the point of view of the parser, which is the input to the parser. The lexical analyzer recognizes words based on a regular grammar so that their NON-TERMINALS become the input elements of later phases.

Syntactic analysis

determines if the sequence of lexical components follows the syntax of the language and obtains the hierarchical structure of the program in the form of a tree, where the nodes are the high-level constructs of the language.
The structural relationships between the lexical components are determined, this is similar to carrying out the grammatical analysis on a phase in natural language.
Syntactic structure is defined by context-independent grammars

. What is the parser?
It is the analyzer phase that is in charge of checking the input text based on a given grammar. And in case the input program is valid, it supplies the syntactic tree that recognizes it. In theory, the output of the parser is assumed to be some representation of the parser that recognizes the sequence of tokens supplied by the parser. In practice, the parser also does:
• Access the symbol table (to do part of the work of the semantic parser).
• Type checking (from the semantic analyzer).
• Generate intermediate code.

Semantic Analysis Semantic
analysis gives a coherent meaning to what we have done in syntactic analysis.
The semantic check ensures that the types that intervene in the expressions are compatible or that the real parameters of a function are consistent with the formal parameters

Main functions
Identify each type of instruction and its components
Complete the Symbol Table
Perform different checks and validations :
Type checks.
Control flow checks.
Uniqueness checks.

Example of a Lexicon-Syntactic-Semantic Analyzer:

<? Xml version = "1.0" encoding = "UTF-8"?>
<! - You may freely edit this file. See commented blocks below for ->
<! - some examples of how to customize the build. ->
<! - (If you delete it and reopen the project it will be recreated.) ->
<! - By default, only the Clean and Build commands use this build script. ->
<! - Commands such as Run, Debug, and Test only use this build script if ->
<! - the Compile on Save feature is turned off for the project. ->
<! - You can turn off the Compile on Save (or Deploy on Save) setting ->

```
<! - in the project's Project Properties dialog box .-->
<project name = "Final Compiler" default = " default "basedir =". ">
    <description> Builds, tests, and runs the project Final Compiler. </description>
    <import file =" nbproject / build-impl.xml "/>
    <! -
    There exist several targets which are by default empty and which can be
    used for execution of your tasks. These targets are usually executed
    before and after some main targets. They are:
      -pre-init: called before initialization of project properties
      -post-init: called after initialization of project properties
      -pre-compile: called before javac compilation
      -post-compile: called after javac compilation
      -pre-compile-single: called before javac compilation of single file
      -post-compile-single: called after javac compilation of single file
      -pre-compile-test: called before javac compilation of JUnit tests
      -post-compile-test: called after javac compilation of JUnit tests
      - pre-compile-test-single: called before javac compilation of single JUnit test
      -post-compile-test-single: called after javac compilation of single JUunit test
      -pre-jar: called before JAR building
      -post-jar: called after JAR building
      -post-clean: called after cleaning build products
    (Targets beginning with '-' are not intended to be called on their own.)
    Example of inserting an obfuscator after compilation could look like this:
        <target name = "- post-compile ">
            <obfuscate>
                <fileset dir =" $ {build.classes.dir } "/>
            </obfuscate>
        </target>
    For list of available properties check the imported
    nbproject / build-impl.xml file.
    Another way to customize the build is by overriding existing main targets.
    The targets of interest are:
      -init-macrodef-javac: defines macro for javac compilation
      -init-macrodef-junit: defines macro for junit execution
      -init-macrodef-debug: defines macro for class debugging
      -init-macrodef-java: defines macro for class execution
      -do-jar: JAR building
      run: execution of project
      -javadoc-build: Javadoc generation
      test-report: JUnit report generation
    An example of overriding the target for project execution could look like this:
```

```
        <target name = "run "depends =" CompilerFinal-impl.jar ">
            <exec dir =" bin "executable =" launcher.exe ">
                <arg file =" $ {dist.jar} "/>
            </exec>
        </target>
    Notice that the overridden target depends on the jar target and not only on
    the compile target as the regular run target does. Again, for a list of available
    properties which you can use, check the target you are overriding in the
    nbproject / build-impl.xml file.
    ->
</project>
```

Frontend example:

```
project-name /
| __ src
|         | __ scss
|         |         | __ style.scss
|         |         | __ inc
|         |                 | __ mixins.scss
|         |                 | __ normalize.scss
|         |                 | __ colors.scss
|         |                 | __ variables.scss
|         |                 | __ components.scss
|         |
|         | __ jade
|         |         | __ page.jade
|         |         | __ inc
|         |         |         | __ mixins.jade
|         |         |         | __ variables.jade
|         |         | __ template
|         |                 | __ templatename.jade
|         |
|         | __ js
|         |         | __ functions.js
|         |
|         | __images
|                 | __ sprites
|
| __ gruntfile.js
| __ package.json
```

```
| __ bower.json
| __ .editorconfig
| __ .gitignore
| __ .htmlhintrc
| __ .jshintrc
|
| __ dist
|       | __ page.html
|       | __ assets
|             | __ css
|             |       | __ style.css
|             |       | __ libs
|             |             | __ anyexternallib.css
|             | __ js
|             |       | __ functions.js
|             |       | __ libs
|             |             | __ jquery-1.11.3.min.js
|             |             | __ modernizr.js
|             |             | __ detectizr.js
|             |             | __ lt-ie-9.min.js
|             |             | __ anyexternallib.js
|             |
|             | __ images
|                   | __ sprites.png
|
| __ build
      | __ page.html
      | __ assets
            | __ css
            |       | __ styles.min.css
            | __ js
            |       | __ functions.min.js
            |       | __ libs
            |             | __ jquery-1.11.3.min.js
            |             | __ modernizr-detectizr.min.js
            |             | __ ie.min.js
            |             | __ plugins.min.js
            |
            | __ images
                  | __ sprites.png
```

Santana Gonzalez Jesus Salvador

3.2.2 "Context-Free Grammars"

A grammar G = hΣ, Γ, S, → i is context-independent if all production rules have one of the following two forms

A → α A →,

where A ∈ Γ.

CFG will designate the set of context-independent grammars.

The language generated by G will be denoted by L (G).

A language L is context-independent if there exists a grammar G ∈ CFG such that

L = L (G).

CFL is the set of context-independent

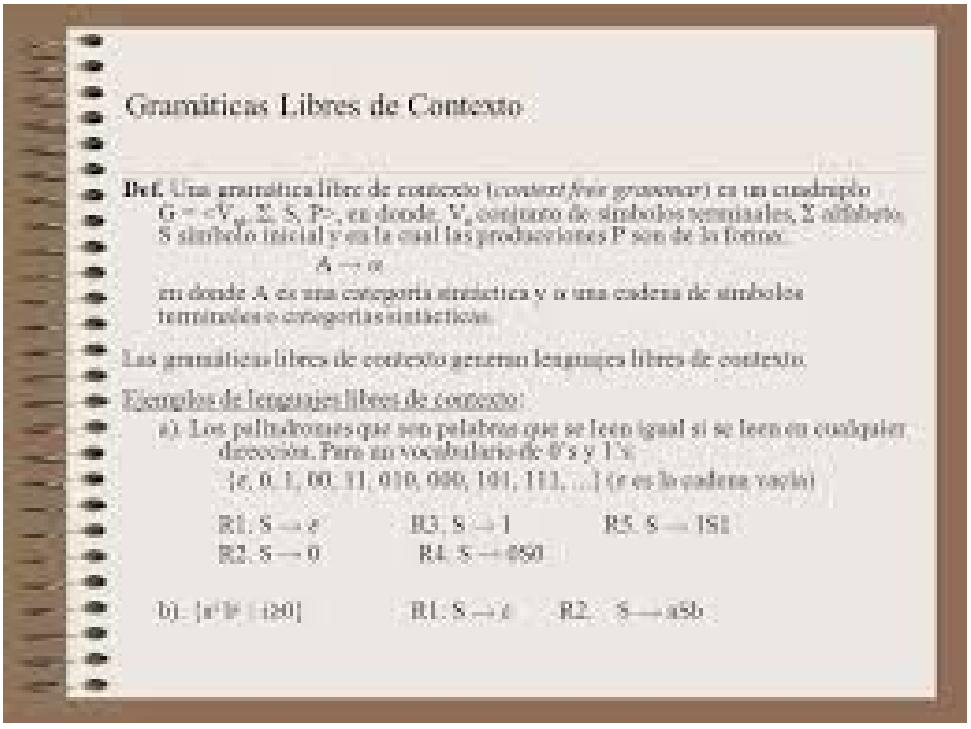
languages Palindrome language. Let GP = h {a, b}, {S}, S, → Pi, where

S → P aSa | bSb | .

2 Language of parentheses. Let GD = h {(,)}, {S}, S, → Di, with the following rules

S → D (S) | SS | .

In formal language theory, a context-free grammar (CFG) is a formal grammar in which each production rule has the form where is a single non-terminal symbol and is a string of terminals and / or non-terminals (it can be empty). A formal grammar is considered "context-free" when its production rules can be applied independently of the context of a nonterminal. Regardless of the surrounding symbols, the single nonterminal on the left side can always be replaced by the right side. This is what distinguishes it from a context-sensitive grammar. A formal grammar is essentially a set of production rules that describe all possible strings in a given formal language. The production rules are simple replacements. For example, the first rule of the image, replace with. There can be multiple replacement rules for a given non-terminal symbol. The language generated by a grammar is the set of all strings of terminal symbols that can be derived, by repeated applications of rules, from some particular non-terminal symbol ("start symbol"). Nonterminal symbols are used during the derivation process, but may not appear in your final result string.



A regular grammar is a 4-tuple G = (Σ, N, S, P), where Σ is the alphabet, N is a collection of non-terminal symbols, S is a non-terminal symbol called the initial symbol, and P is a set of substitution rules, called productions of the form A → w where A∈N and w∈ (Σ∪N) satisfying:
1) w contains at most one nonterminal.
2) If w contains a nonterminal, then it is the symbol at the far right of w. z

Santana Gonzalez Jesus Salvador

Language processors

Programming languages are notations that describe calculations to people and machines. Our perception of the world we live in depends on programming languages, since all software that runs on all computers was written in some programming language. The software systems that handle this translation are called compilers. A compiler is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language). The target machine language program produced by a compiler is generally faster than an interpreter in mapping inputs to outputs.

The structure of a compiler

The analysis part divides the source program into components and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. The analysis part 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 parsing part is commonly called the compiler front-end; the part of the synthesis (properly the translation) is the back-end. Some compilers have a machine-independent code optimization phase, between the front-end and the back-end. The purpose of this optimization is to perform transformations on the intermediate representation, so that the back-end can produce a better target program than it would have produced with a non-optimized intermediate representation.

Lexical AnalysisLexical

TheAnalyzer reads the stream of characters that make up the source program and groups them into meaningful sequences, known as lexemes.

Parsing

The parser (parser) uses the first few token components produced by the lexical parser to create an intermediate representation in the form of a tree that describes the grammatical structure of the token stream. A typical representation is the syntactic tree, in which each interior node represents an operation and the children of the node represent the arguments of the operation.

The following stages of the compiler use the grammatical structure to help parse the source program and generate the target program.

Semantic analysis

Uses the syntactic tree and the information in the symbol table to check the semantic consistency of the source program with the language definition. It also collects information about the type and saves it, either to the syntax tree or to the symbol table, for later use

during the generation of intermediate code. An important part of semantic analysis is type checking, where the compiler verifies that each operator has matching operands.

("Lexical Analysis (parts 2 and 3)")

Binary translation
Compiler technology can be used to translate binary code for one machine into binary code for a different machine, thereby allowing a machine to execute programs that were originally compiled for another set of instructions. Binary translation can also be used for backward compatibility.

Hardware Synthesis
Not only is most software written in high-level languages, even most hardware designs are described in high-level hardware description languages, such as Verilog and VHDL (Very High-Speed Integrated Circuit Hardware Description Language)

Compiled
simulation Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon, or to validate a design. or general, the simulator inputs include the design description and the specific input parameters for that specific simulation run. Simulations can be very expensive. Generally, we need to simulate many possible design alternatives on many different input sets, and each experiment can take days to complete, on a high-performance machine.

When talking about lexical analysis, we use three different, but related terms:
A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol that represents a type of lexical unit; for example, a specific keyword or a sequence of input characters denoting an identifier. The token names are the input symbols that are processed by the parser. From now on, we will generally write the name of a token in bold. We will often refer to a token by name.

A pattern is a description of the form that the lexemes of a token can take. In the case of a keyword like token, the pattern is just the sequence of characters that make up the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is related through many strings.

A lexeme is a sequence of characters in the source program, matching the pattern for a token, and identified by the lexical analyzer as an instance of that token.

Attributes for tokens

When more than one lexeme can match a pattern, the parser must provide subsequent compiler stages with additional information about the specific lexeme that matched.

Other functions it performs:
• Delete comments from the program.
• Eliminate blank spaces, tabs, carriage return, etc., and in general, everything that lacks meaning according to the syntax of the language.
• Recognize user identifiers, numbers, reserved words of the language, ..., and treat them correctly with respect to the symbol table (only in cases where you should deal with the symbol table).
• Keep track of the line number you are reading from, in case an error occurs, give information about where it occurred.

Lexical Errors
Without the help of the other components it is difficult for a Lexical Analyzer to know that there is an error in the source code. The simplest recovery strategy is "panic mode" recovery. We remove successive characters from the rest of the input, until the lexical analyzer can find a well-formed token at the beginning of what is left of the input. This retrieval technique can confuse the parser, but in an interactive computing environment, it can be quite suitable. Other possible error recovery actions are:

1. Remove a character from the rest of the input.
2. Insert a missing character in the rest of the entry.
3. Replace one character with another.
4. Transpose two adjacent characters.

Transformations like these can be tested in an attempt to repair the input. The simplest strategy is to see if a prefix from the rest of the input can be transformed into a valid lexeme by a simple transformation. This strategy makes sense, since in practice most lexical errors involve only one character. A more general correction strategy is to find the fewest number of transformations necessary to convert the source program to one consisting of only valid lexemes, but this method is considered too expensive in practice to be worth doing.

Buffering input
Before we talk about the problem of recognizing lexemes in input, let's examine some ways in which the simple but important task of reading the source program can be streamlined. This task is made difficult because we often have to look for one or more characters beyond the next lexeme in order to be sure that we have the correct lexeme. In C, single-character operators like -, =, or <could also be the beginning of a two-character operator, like ->, ==, or <=. Hence, we are going to introduce a two-buffer

scheme that handles long read-aheads with no problems. Later we will consider an enhancement in which “sentries” are used to save time when checking the end of buffers.

Buffer Pairs

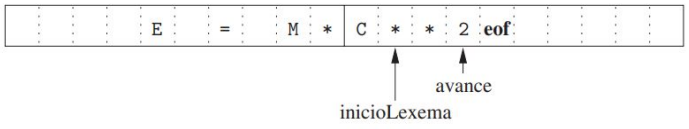
Due to the time required to process characters and the large number of characters that must be processed during compilation of a large source program, specialized buffer usage techniques have been developed to reduce the amount of overhead required in processing a single input character.

Each buffer is the same size N, and typically N is the size of a disk block (that is, 4,096 bytes). By using a system read command we can read N characters and put them in a buffer, instead of using a system call for each character. If there are fewer than N characters left in the input file, then a special character, represented by eof, marks the end of the source file and is different from any of the possible characters in the source program. Two pointers are kept to the input:

1. The startLexema pointer marks the beginning of the current lexeme, whose extension we are trying to determine.
2. The forward pointer scans ahead until it finds a match in the pattern; During the remainder of the chapter we will cover the exact strategy by which this determination is made.

Once the next lexeme is determined, forward is placed on the character at its far right. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, startLexeme is placed on the character that comes just after the lexeme we just found.

Sentinels



If we use the scheme in the way described, we must verify, each time we move the forward pointer, that we have not left one of the buffers; if this happens, then we must also reload the other buffer. Thus, for each character reading we do two tests: one for the end of the buffer and the other to determine which character is read (the latter can be a multi-way fork). We can combine the end of buffer test with the current lack test by extending each buffer to contain a sentinel value at the end. The sentinel is a special character that cannot be part of the source program, for which a natural choice is the eof character.

Strings and languages

An alphabet is a finite set of symbols. Some typical examples of symbols are letters, digits, and punctuation marks. The set {0, 1} is the binary alphabet. ASCII is an important example of an alphabet; it is used in many software systems.

A string over an alphabet is a finite sequence of symbols that are drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used synonymously with "string." The length of a string s, usually written as | s |, is the number of symbol occurrences in s.

Operations in languages

In lexical analysis, the most important operations in languages are union, concatenation, and lock

OPERACIÓN	DEFINICIÓN Y NOTACIÓN
Unión de L y M	$L \cup M = \{\ s \mid s$ está en L o s está en $M\ \}$
Concatenación de L y M	$LM = \{st \mid s$ está en L y t está en $M\ \}$
Cerradura de Kleene de L	$L^* = \mathrm{U}_{i=0}^{\infty} L^i$
Cerradura positivo de L	$L^+ = \mathrm{U}_{i=1}^{\infty} L^i$

Figura 3.6: Definiciones de las operaciones en los lenguajes

Regular expressions

It is used to describe all the languages that can be built from these operators, applied to the symbols of a certain alphabet. In this notation, if letter_ is set to represent any letter or underscore, and digit_ is Figure 3.6: Definitions of operations in languages OPERATION DEFINITION AND NOTATION Union of L and ML ∪ M = {s | s is in L os is in M} Concatenation of L and M LM = {st | s is in L and t is in M} Kleene lock of LL * = U∞ i = 0 Li Positive lock of L L + = U∞ i = 1 Li Maq. Cap_3_AHO.indd 120 11/10/07 12:48:47 AM set to represent any digit, then we could describe the language of C identifiers using the following:

letter_ (letter_ | digit) *

The vertical bar of the previous expression means the union, the parentheses are used to group the subexpressions, the asterisk indicates “zero or more occurrences of”, and the juxtaposition of letter_ with the rest of the expression indicates concatenation. Regular expressions are built recursively from smaller regular expressions, using the rules we will describe below. Each regular expression r denotes a language L (r), which is also defined recursively, from the languages denoted by the subexpressions of r. Here are the rules that define regular expressions on a certain alphabet Σ, and the languages that denote those expressions.

BASE: There are two rules that form the base:

1. is a regular expression, and L () is {}; that is, the language whose only member is the empty string.
2. If a is a symbol in Σ, then a is a regular expression, and L (a) = {a}, that is, the language with a string, of length one, with a in its only position. Note that by convention we use italics for symbols, and bold for their corresponding regular expression.

INDUCTION: There are four parts to induction, whereby the largest regular expressions are constructed from the smallest. Suppose r and s are regular expressions denoting the languages L (r) and L (s), respectively.

1. (r) | (s) is a regular expression denoting the language L (r) ∪ L (s).
2. (r) (s) is a regular expression denoting the language L (r) L (s).
3. (r) * is a regular expression denoting (L (r)) *.
4. (r) is a regular expression that denotes L (r). This last rule says that we can add additional pairs of parentheses around expressions, without changing the language they denote.

By definition, regular expressions often contain unnecessary pairs of parentheses. It may be necessary to remove certain pairs of parentheses, if we adopt the following conventions:

a) The unary operator * has the highest precedence and is left-associative.
b) The concatenation has the second highest precedence and is left associative.
c) | has the lowest precedence and is left associative.

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and a denote the same regular set, we say that they are equivalent and write r = s. For example, (a | b) = (b | a). There are a variety of algebraic laws for regular expressions; Each law states that expressions of two different forms are equivalent. Figure 3.7 shows part of the algebraic laws for the arbitrary regular expressions r, s and t.

LEY	DESCRIPCIÓN
$r\|s = s\|r$	$\|$ es conmutativo
$r\|(s\|t) = (r\|s)\|t$	$\|$ es asociativo
$r(st) = (rs)t$	La concatenación es asociativa
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	La concatenación se distribuye sobre $\|$
$\epsilon r = r\epsilon = r$	ϵ es la identidad para la concatenación
$r^* = (r\|\epsilon)^*$	ϵ se garantiza en un cerradura
$r^{**} = r^*$	* es idempotente

Extensions to regular expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene's lock in the 1950s, many extensions have been added to regular expressions to improve their ability to specify patterns. of chains.

1. One or more instances. The unary postfix operator + represents the positive closure of a regular expression and its language. That is, if r is a regular expression, then (r) + denotes the language (L (r)) +. The + operator has the same precedence and associativity as the * operator. Two useful algebraic laws, r * = r + | and r + = rr * = r * r relate the Kleene lock and the positive lock.

2. Zero or one instance. The postfix unary operator? means "zero or one occurrence". That is, r? is equivalent to r |, or put another way, L (r?) = L (r) ∪ {}. The operator ? has the same precedence and associativity as * and +.

3. Character classes. A regular expression a1 | a2 | · · · | an, where the ais are each alphabet symbols, can be replaced by the abbreviation [a1a2 ··· an]. Most importantly, when a1, a2, ..., an form a logical sequence, for example uppercase letters, lowercase letters or consecutive digits, we can substitute a1-an for them; that is, only the first and last separated by an underscore. Thus, [abc] is the abbreviation for a | b | c, and [az] is it for a | b | · · · | z.

Token recognition

For oprel, we use the comparison operators of languages such as Pascal or SQL, where = is “is equal to” and <> is “is not equal to”, since it presents an interesting structure of lexemes. The terminals of the grammar, which are if, then, else, oprel, id and number, are the names of tokens as far as the lexical analyzer is concerned.

$$
\begin{array}{rcl}
\mathit{digito} & \rightarrow & \texttt{[0-9]} \\
\mathit{digitos} & \rightarrow & \mathit{digito}^{+} \\
\mathit{numero} & \rightarrow & \mathit{digitos}\ (\texttt{.}\ \mathit{digitos})?\ (\ \texttt{E}\ [\texttt{+-}]?\ \mathit{digitos}\ )? \\
\mathit{letra} & \rightarrow & \texttt{[A-Za-z]} \\
\mathit{id} & \rightarrow & \mathit{letra}\ (\ \mathit{letra}\ |\ \mathit{digito}\ )^{*} \\
\mathit{if} & \rightarrow & \texttt{if} \\
\mathit{then} & \rightarrow & \texttt{then} \\
\mathit{else} & \rightarrow & \texttt{else} \\
\mathit{oprel} & \rightarrow & \texttt{<}\ |\ \texttt{>}\ |\ \texttt{<=}\ |\ \texttt{>=}\ |\ \texttt{=}\ |\ \texttt{<>}
\end{array}
$$

For this language, the lexical analyzer will recognize the keywords if, then, and else, as well as the lexemes that match the patterns for oprel, id, and number. To keep things simple, let's make the common assumption that keywords are also reserved words; that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

In addition, we will assign the lexical analyzer the work of eliminating the white space, recognizing the "token" is defined by:

ws → (blank | tab | nuevalinea) +

Here, blank, tab and nuevalinea are abstract symbols that we use to express the characters ASCII of the same names. The ws token is different from the other tokens because when we recognize it, we do not return it to the parser, but we restart the parser from the character that comes after the white space. The next token is the one returned to the parser.

LEXEMAS	NOMBRE DEL TOKEN	VALOR DEL ATRIBUTO
Cualquier *ws*	–	–
`if`	**if**	–
`Then`	**then**	–
`else`	**else**	–
Cualquier *id*	**id**	Apuntador a una entrada en la tabla
Cualquier *numero*	**numero**	Apuntador a una entrada en la tabla
`<`	**oprel**	`LT`
`<=`	**oprel**	`LE`
`=`	**oprel**	`EQ`
`<>`	**oprel**	`NE`
`>`	**oprel**	`GT`
`>=`	**oprel**	`GE`

State transition diagrams

As an intermediate step in building a lexical analyzer, we first convert the patterns into stylized flow diagrams, which are called "state transition diagrams".
State transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that might occur during the process of exploring the input, looking for a lexeme that matches one of several patterns. We can consider a state as a summary of everything we need to know about the characters that we have seen between the startLexema pointer and the forward pointer.

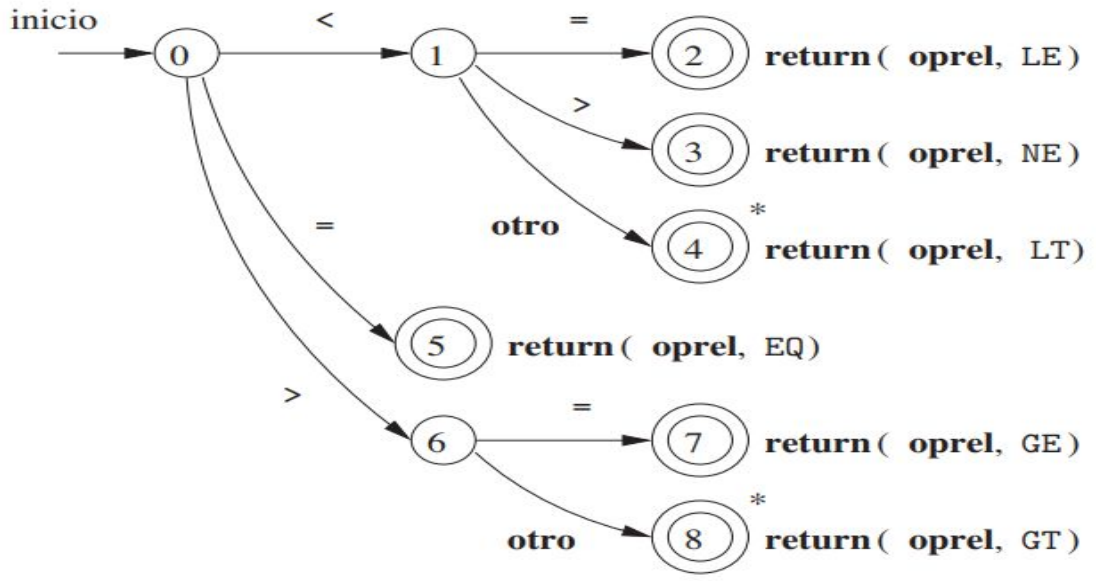
The lines are directed from one state to another on the state transition diagram. Each line is labeled by a symbol or set of symbols. If we are in a certain state s, and the next input symbol is a, we look for a line that comes out of state s and is labeled by a (and perhaps other symbols as well). If we find this line, we advance the forward pointer and enter the state of the state transition diagram to which that line takes us. We will assume that all of our state transition diagrams are deterministic, which means that there is never more than one line coming out of a given state, with a given symbol between its labels.

Some important conventions of state transition diagrams are:

1. Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the current lexeme may not consist of all positions between the startLexeme and forward pointers. We always indicate an accept state by a double circle, and if an action needs to be taken (usually returning a token and an attribute value to the parser), we will attach it to the accept state.

2. Also, if it is necessary to move the pointer back one position (that is, if the lexeme does not include the symbol that brought us to the accepting state), then we must additionally place a * near the accepting state.

3. A state is designated as the initial state; this is indicated by a line labeled "start", which does not come from anywhere. The transition diagram always starts in the initial state, before reading any input symbols.

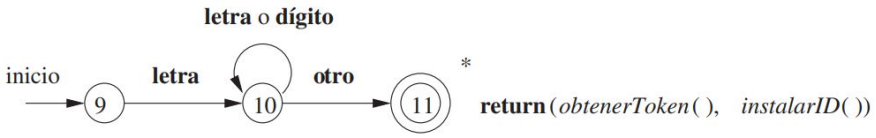


On the other hand, if in state 0 the first character we see is =, then this character must be the lexeme. We immediately return that fact from state 5. The remaining possibility is that the first character is>. Then, we must go to state 6 and decide, based on the next character, if the lexeme is> = (if we see the = sign below), or only> (with any other character). Note that, if in state 0 we see any character other than <, = or>, it is not possible that we are seeing an oprel lexeme, so we will not use this state transition diagram.

Recognition of reserved words and identifiers

Recognition of reserved words and identifiers presents a problem. Keywords like if or then are generally reserved (as in our sketch), so they are not identifiers, even though they appear to be. Thus, although we generally use a state transition diagram to look up

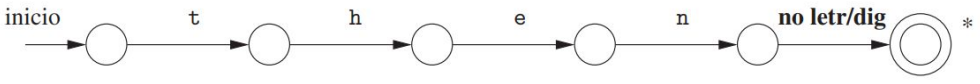
identifier lexemes, this diagram will also recognize the if, then, and else keywords from our sketch.



There are two ways we can handle reserved words that look like identifiers:

1. Install the reserved words in the symbol table from scratch. A field in the symbol table entry indicates that these strings will never be ordinary identifiers, and tells us which token they represent. Upon finding an identifier, a call to installID places it in the symbol table, if it is not already there, and returns a pointer to the entry in the symbol table for the lexeme that was found. The getToken function examines the entry in the token table for the found lexeme, and returns the token name that the token table indicates this lexeme represents; either id or one of the keyword tokens that was originally installed on the table.

2. Create separate state transition diagrams for each keyword. If we adopt this method, then we must give priority to the tokens, so that the reserved word tokens are recognized in preference instead of id, when the lexeme matches both patterns.



Sketch completion

The state transition diagram for token number is shown in Figure 3.16, and it is the most complex diagram we have seen so far. Starting in state 12, if we see a digit we go to state 13. In that state we can read any number of additional digits. However, if we see something other than a digit or a point, we have seen a number as an integer; 123 is an example. To handle this case, we go to state 20, where we return the token number and a pointer to a table of constants where the found lexeme is entered. This mechanic is not shown in the diagram, but it is analogous to the way we handle identifiers.

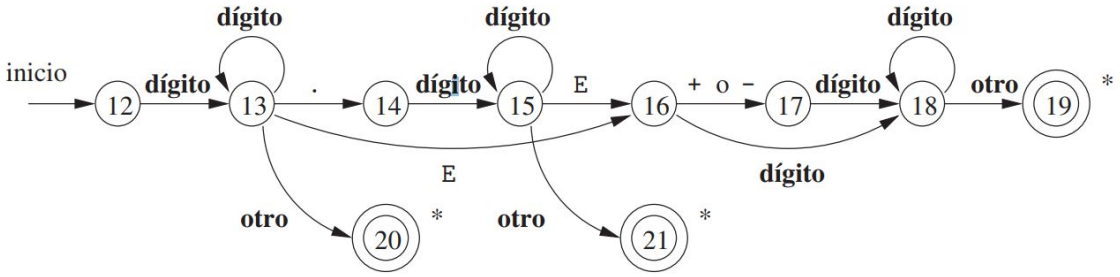


Figura 3.16: Un diagrama de transición para los números sin signo

If we instead see a point in state 13, then we have an "optional fraction". We go to state 14, and we look for one or more additional digits; state 15 is used for this purpose. If we see an E, then we have an “optional exponent”, whose recognition is the work of states 16 to 19. If in state 15 we see something other than an E or a digit, then we have reached the end of the fraction, not there is an exponent and we return the found lexeme, using state 21.

The final state transition diagram, shown in Figure 3.17, is for the blank space. In that diagram we look for one or more "white space" characters, represented by delim in that diagram; generally these characters are spaces, tabs, newline characters, and perhaps other characters that the language design does not consider to be part of some token.

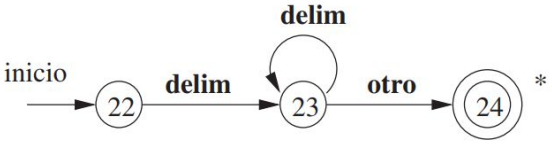


Figura 3.17: Un diagrama de transición para el espacio en blanco

Notice that, in state 24, we found a block of consecutive whitespace characters, followed by a non-whitespace character. Let's return the input to start at the non-whitespace character, but we don't return anything to the parser, instead we must restart the parsing process after the whitespace.

Architecture of a lexical analyzer based on state transition diagrams

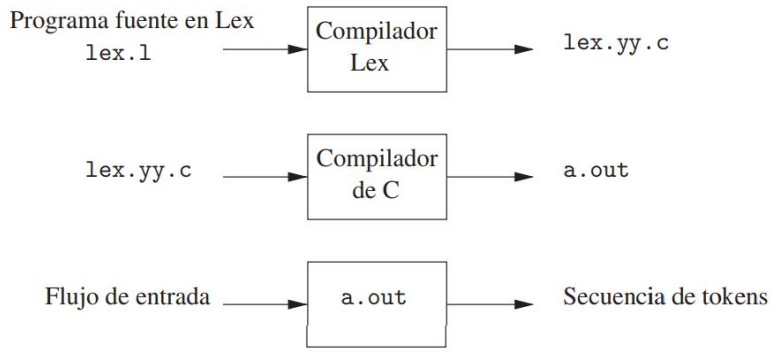
A switch instruction based on the state value takes us to the code for each of the possible states, where we find the action of that state. Often times, the code for a state is itself a switch statement or a multi-way branch that determines the next state through the process of reading and examining the next input character.

The Lex parser generator

allows us to specify a lexical parser by specifying regular expressions to describe token patterns. The input notation for the Lex tool is known as the Lex language, and the tool itself is the Lex compiler. The Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram. The mechanics of how this translation from regular expressions to transition diagrams occurs is the subject of the following sections; here we will only learn about the Lex language.

Using Lex

An input file, which we will call lex.l, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms lex.l into a C program, into a file that is always called lex.yy.c. The C compiler compiles this file into a file called a.out, as usual. The output of the C compiler is a functional lexical analyzer, which can receive a stream of input characters and produce a string of tokens.



Structure of LexLex

programs Aprogram has the following form:

declarations
%%
translation rules
%%
helper functions

The declarations section includes variable declarations, manifest constants (identifiers that are declared to represent a constant; for example, the name of a token), and regular definitions.

Each of the translation rules has the following form
:

Pattern {Action}

Each pattern is a regular expression, which can use the regular definitions from the declarations section. Actions are snippets of code, typically written in C, although many variants of Lex have been created that use other languages. The third section contains the additional functions that are used in actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer. The lexical parser that Lex creates works in conjunction with the parser as follows. When the parser calls the parser, it starts reading the rest of your input, one character at a time, until it finds the longest prefix of the input that matches one of the Pi patterns. Then it executes the associated action Ai. Usually Ai will return to the parser, but if it doesn't (perhaps because Pi describes whitespace or comments), then the parser proceeds to look for additional lexemes, until one of the corresponding actions causes a return. to the parser. The parser returns a single value, the name of the token, to the parser, but uses the shared integer variable yylval to pass additional information about the found lexeme to it, if necessary.

Conflict resolution in Lex
We have referred to the two rules that Lex uses to decide on the appropriate lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter one.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Theforward operator
Lexautomatically reads one character ahead of the last character that makes up the selected lexeme, and then returns the input so that only the input lexeme itself is consumed. However, sometimes it may be desirable to have a certain pattern match the input, only when it is followed by certain other characters. If so, perhaps we can use the forward slash in a pattern to indicate the end of the part of the pattern that matches the lexeme. What comes after / is an additional pattern that must be related before we can decide that we saw the token in question, but that what matches this second pattern is not part of the lexeme.

Finite automata
At the heart of the transition is the formalism known as finite automata. In essence, these consist of graphs like state transition diagrams, with a few differences:

1. Finite automata are recognizers; they just say "yes" or "no" in relation to each possible input string.

2. Finite automata can be of two types:

(a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their lines. A symbol can label multiple lines arising from the same state, and the empty string is a possible label.

(b) Deterministic finite automata (AFD) have, for each state, and for each symbol of their input alphabet, exactly one line with that symbol coming out of that state.

Both deterministic and non-deterministic finite automata are capable of recognizing the same languages. In fact, these languages are exactly the same languages, known as regular languages, that can describe regular expressions.

Non-deterministicnon-deterministic
finite automata Afinite automata (NFA) consists of:

1. A finite set of states S.

2. A set of input symbols Σ, the input alphabet. We assume that, representing the empty string, it will never be a member of Σ.

3. A transition function that provides, for each state and for each symbol in Σ ∪ {}, a set of subsequent states.

4. A state s0 from S, distinguished as the initial state.

5. A set of states F, a subset of S, which are distinguished as the accepting states (or final states).

We can represent an AFN or AFD by means of a transition graph, where the nodes aren states and the Undecidable edges represent the transition function. There is an Undecidable edge a, which goes from state s to state t if, and only if t is one of the following states for state s and input a. This graph is very similar to a transition diagram, except that:

a) The same symbol can label edges from one state to several different states.

b) An edge can be labeled by, the empty string, instead of, or in addition to, the input alphabet symbols.

Transition tables

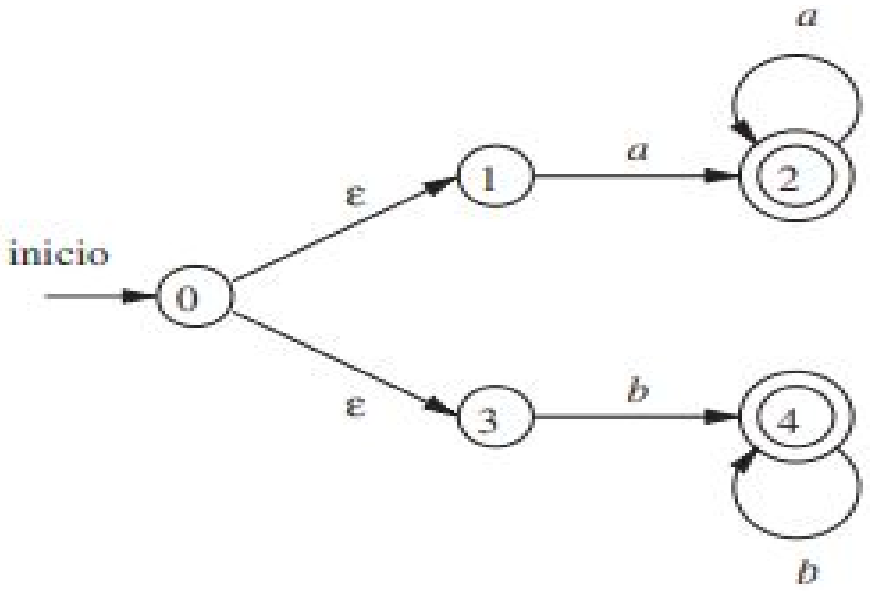
We can also represent an AFN by means of a transition table, whose rows correspond to the states, and whose columns correspond to the input symbols already ε . The input for a given state and the input is the value of the transition function that applies to those arguments. If the transition function has no information about that state-input pair, we place ∅ in the table for that state.

Acceptance of input strings by automata

An AFN accepts the input string x if, and only if, there is a path in the transition graph, from the initial state to one of the acceptance states, so that the symbols along along the way spell x. Note that the tags ε along the path are ignored as the empty string does not contribute to the chain that is built along the path.

Deterministic finite

automata A deterministic finite automata (PDA) is a special case of an AFN, where:



1. There are no movements at the input ε.

2. For each state s and each input symbol a, there is exactly one line arising from s, Undecidable as a.

If we use a transition table to represent an AFD, then each input is a single state. Therefore, we can represent this state without the keys that we use to form the sets.

While the AFN is an abstract representation of an algorithm to recognize the strings of a certain language, the AFD is a simple and concrete algorithm to recognize strings. It is certainly fortunate that each regex and each AFN can become an AFD that accepts the same language, as it is the AFD that we actually implement or simulate when building lexical analyzers. The following algorithm shows how to apply an AFD to a string.

Converting an AFN to AFD

The general idea of the construction of subsets is that each state of the constructed AFD corresponds to a set of states of the AFN. After reading the input a1a2 ··· an, the AFD is in the state that corresponds to the set of states that the AFN can reach, from its initial state, following the paths labeled a1a2 ··· an. It is possible that the number of AFD states is exponential in the number of AFN states, which could cause difficulties when trying to implement this AFD. However, part of the power of the automaton-based method for lexical analysis is that for real languages, the AFN and the AFD have approximately the same number of states, and the exponential behavior is not seen.

Algorithm 3.20: The construction of subsets of an AFD, from an AFN.
INPUT: An AFN N.
OUTPUT: An AFD D that accepts the same language as N.
METHOD: Our algorithm constructs a transition table Dtran for D. Each state of D is a set of states of the AFN, and we construct Dtran so that D can simulate "in parallel" all the possible movements that N can perform on a given input string. Our first problem is to handle the transitions of N properly. In figure 3.31 we see the definitions of several functions that describe basic calculations in the states of N that are necessary in the algorithm. Note that s is an individual state of N, while T is a set of states of N.

OPERACIÓN	DESCRIPCIÓN
ϵ-*cerradura*(s)	Conjunto de estados del AFN a los que se puede llegar desde el estado s del AFN, sólo en las transiciones ϵ.
ϵ-*cerradura*(T)	Conjunto de estados del AFN a los que se puede llegar desde cierto estado s del AFN en el conjunto T, sólo en las transiciones ϵ; $= \cup_{s \text{ en } T}$ ϵ-*cerradura*(s).
mover(T, a)	Conjunto de estados del AFN para los cuales hay una transición sobre el símbolo de entrada a, a partir de cierto estado s en T.

Figura 3.31: Operaciones sobre los estados del AFN

We must explore those sets of states that N can be in after looking at a certain input string. As a basis, before reading the first input symbol, N can be in any of the -lock states (s0), where s0 is its initial state. For induction, suppose that N can be in the set of states T after reading the input string x. If he next reads the input a, then N can immediately go to any of the states in move (T, a). However, after reading a, you could also make several transitions; therefore N could be in any -lock state (move (T, a)) after reading input xa. Following these ideas, the construction of the set of states of D, Unbound, and its transition function Dtran.

The structure of the generated parser

Figure 3.49 presents the generalities about the architecture of a lexical parser generated by Lex. The program that serves as a lexical analyzer includes a fixed program that simulates

an automaton; at this point we leave open the decision of whether the automaton is deterministic or not. The rest of the lexical analyzer consists of components that are created from the Lex program, by Lex himself.

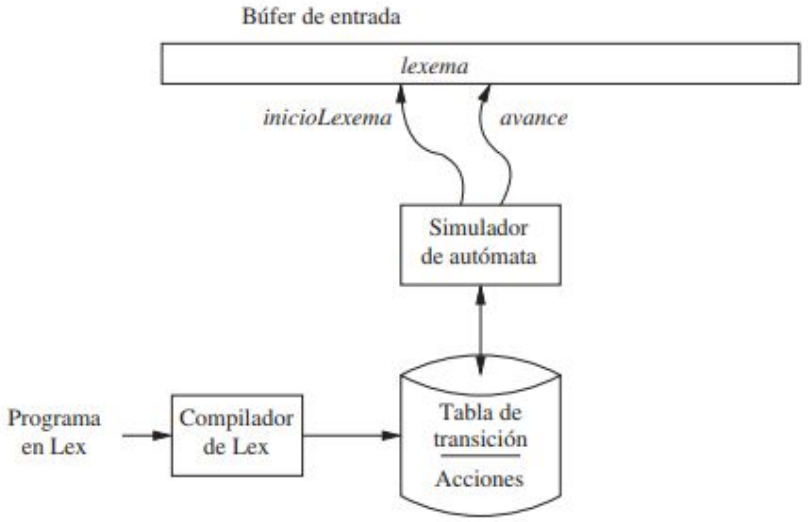


Figura 3.49: Un programa en Lex se convierte en una tabla de transición y en acciones, para que las utilice un simulador de autómatas finitos

Pattern matching based on AFNs

If the lexical analyzer simulates an AFN like the one in Figure 3.52, then it should read the input that starts at the point of its input, which we have referred to as startLexema. As the pointer called forward moves forward through the input, it computes the set of states it is in at each point.

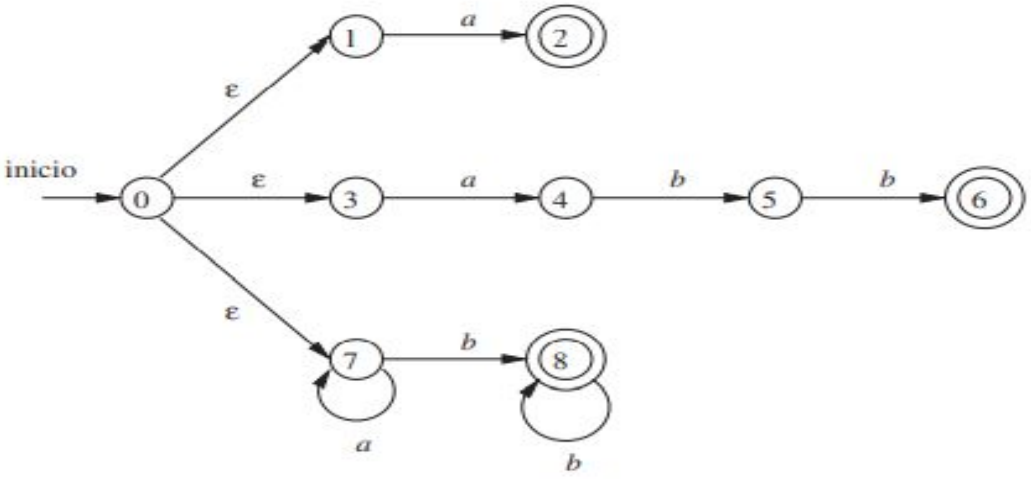


Figura 3.52: AFN combinado

At some point, the AFN simulation reaches a point at the input where there are no following states. At that point, there is no hope that any longer prefix of the input will cause the AFN to

go into an accepting state; instead, the set of states will always be empty. Therefore, we are ready to decide on the longest prefix that is a lexeme that matches a certain pattern.

We search backward through the sequence of sets of states, until we find a set that includes one or more accepting states. If there are multiple acceptance states in that set, we choose the one associated with the first pi pattern in the Lex program list. We move back the forward pointer towards the end of the lexeme, and perform the action Ai associated with the pattern pi.

AFDs for lexical analyzers

Within each AFD state, if there are one or more accepting states of the AFN, the first pattern whose accepting state is represented is determined, and that pattern becomes the output of the AFD state.

Implementing the Look-ahead Operator

When converting the r1 / r2 pattern to an AFN, we treat the / as if it were, so we don't actually look for a / in the input. However, if the AFN recognizes an xy prefix from the input buffer, such that it matches this regular expression, the end of the lexeme is not where the AFN entered its accepting state. Instead, the ending occurs when the NFA enters a state s such that:

1. s has a transition in the imaginary /.

2. There is a path from the initial state of the AFN to state s, which spells x.

3. There is a path from state s to state of acceptance that spells out y.

4. x is as long as possible for any xy that meets conditions 1-3.

If there is only one transition state in the imaginary / in the AFN, then the end of the lexeme occurs when this state is entered for the last time, as illustrated in the following example. If the NFA has more than one transition state in the imaginary /, then the general problem of finding the current state s becomes much more difficult.

AFD-based pattern matching search engine optimization

In this section we will introduce three algorithms used to implement and optimize pattern matching search engines, built from regular expressions.

1. The first algorithm is useful in a Lex compiler, as it builds an AFD directly from a regular expression, without building an intermediate AFN. Also, the resulting AFD can have fewer states than the AFD that is built using an AFN.

2. The second algorithm minimizes the number of states of any AFD, by combining the states that have the same future behavior. The algorithm itself is quite efficient, since it is executed in a time O (n log n), where n is the number of states of the AFD.

3. The third algorithm produces more compact representations of the transition tables than the standard two-dimensional table.

Significant states of an AFN

During the construction of subsets, two sets of states of the AFN (which are treated as if they were the same set) can be identified if:

1. They have the same significant states.
2. Whether they both have states of acceptance, or neither.

The constructed AFN has only one accept state, but this, which has no exit transitions, is not a significant state. By concatenating a single right trailing # marker with a regular expression r, we provide the accept state for r with a transition over #, thereby marking it as a significant state of the AFN for (r) #. In other words, by using the augmented regular expression (r) #, we can forget about the accept states as subsets construction proceeds; when construction completes, any state with a transition over # must be an accepting state.
The significant states of the AFN correspond directly to the positions in the regular expression that contain alphabet symbols. As we will see soon, it is convenient to present the regular expression through its syntactic tree, where the leaves correspond to the operands and the interior nodes correspond to the operators. An interior node is called concat-node, o-node, or asterisk-node if it is labeled by the concatenation operator (period), the union operator |, or the * operator, respectively.

Functions calculated from the syntactic tree

To build an AFD directly from a regular expression, we build its syntactic tree and then we calculate four functions: nullable, firstpos, lastpos and nextpos, which are defined below. Each definition refers to the syntax tree for a specific augmented regular expression (r) #.

1. nullable (n) is true for a node n of the syntactic tree if, and only if, the subexpression represented by n has a in your language. That is, the subexpression can be “made null” or it can be the empty string, even though it can also represent other strings.

2. firstpos (n) is the set of positions in the subtree rooted in n, which corresponds to the first symbol of at least one string in the language of the subexpression rooted in n.

3. lastpos (n) is the set of positions in the subtree rooted in n, which corresponds to the last symbol of at least one string in the language of the subexpression rooted in n.

4. followingpos (p), for a position p, is the set of positions q in the entire syntactic tree, in such a way that there is a certain string x = a1a2 ··· an in an L ((r) #) such that for certain i, there is a way to explain the membership of x in L ((r) #), making ai coincide with the position p of the syntactic tree and ai + 1 with the position q

Calculation of nullable, firstpos and lastpos

Finally, we need to see how to calculate nextpos. There are only two ways we can make the position of one regex follow another.

1. If n is a concat-node with left child c1 and right child c2, then for each position i in lastpos (c1), all positions in firstpos (c2) are found in nextpos (i).

2. If n is an asterisk-node and i is a position in lastpos (n), then all positions in firstpos (n) are in nextpos (i).

Calculation of nextpos

Finally, we need to see how to calculate nextpos. There are only two ways we can make the position of one regex follow another. 1. If n is a concat-node with left child c1 and right child c2, then for each position i in lastpos (c1), all positions in firstpos (c2) are found in nextpos (i). 2. If n is an asterisk-node and i is a position in lastpos (n), then all positions in firstpos (n) are in nextpos (i).

Direct conversion of a regular expression to an AFD

Algorithm 3.36: Construction of an AFD from a regular expression r.

INPUT: A regular expression r.

OUTPUT: A D AFD that recognizes L (r).

METHOD:

1. Construct a syntactic tree T from the augmented regular expression (r) #.

2. Calculate nullable, first, last and next positions for T, using the methods in sections 3.9.3 and 3.9.4.

3. Construct Destado, the set of states of the AFD D, and Dtran, the transition function for D, using the procedure in figure 3.62. The states of D are states of positions in T. At first, each state is "unmarked", and one state is "marked" just before we consider its exit transitions. The initial state of D is firstpos (n0), where node n0 is the root of T. The accepting states are those that contain the position for the final marker symbol #.

Exchange of time for space in the simulation of an AFD

The simplest and fastest way to represent the transition function of an AFD is a two-dimensional table indexed by states and characters. Given a state and the next input character, we access the array to find the next state and any special actions we need to take; for example, returning a token to the parser. Since an ordinary lexical analyzer has several hundred states in its AFD and involves the ASCII alphabet of 128 input characters, the array consumes less than one megabyte.

However, compilers also appear on very small devices, where up to a megabyte of memory could be too much. For such situations, there are many methods that we can use to compact the transition table. For example, we can represent each state by a list of transitions (that is, character-state pairs) that are terminated by a default state, which must be chosen for any input characters not found in the list. By choosing the next most frequently occurring state as the default, we can often reduce the amount of storage required by a large factor.

There is a more subtle data structure that allows us to combine speed of accessing arrays with compression of lists with default values. We can consider this structure as four arrangements. The base array is used to determine the base location of the entries for state s, which are found in the next and checkout arrays. The default array is used to determine an alternative base location, if the checking array tells us that the one providing base [s] is invalid.

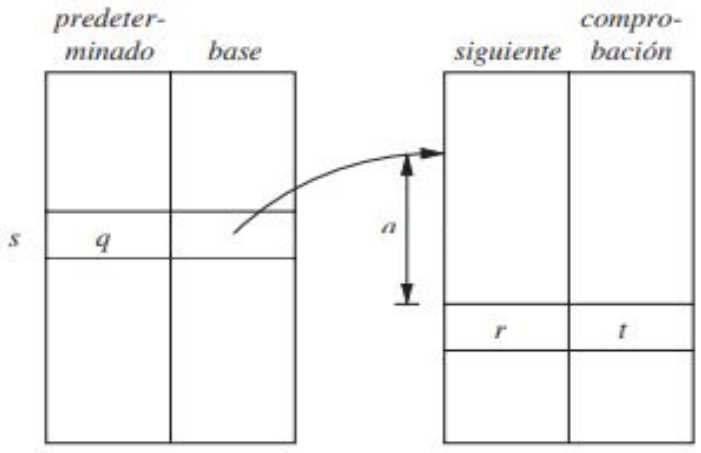


Figura 3.66: Estructura de datos para representar tablas de transición

To calculate nextState (s, a), the transition for state s with input a, we examine the following inputs and check at location l = base [s] + a, where character a is treated as an integer, supposedly in the range from 0 to 127. If check [l] = s, then this input is valid and the next state for state s with input a is next [l]. If check [l] ≠ s, then we determine another state t = default [s] and repeat the process, as if t were the current state. More formally, the followingState function is defined like this:

```
int siguienteEstado(s, a) {
        if ( comprobacion[base[s]+a] = s ) return siguiente[base[s] + a];
        else return siguienteEstado(predeterminado[s], a);
}
```

("Syntactic Analysis (part 1)")
In fact, the parse tree does not need to be explicitly constructed, since the check and translate actions can be interspersed with the analysis syntactic, as we will see later. Hence, the parser and the rest of the user interface could be seamlessly implemented using a single module.

A grammar provides a precise, yet easy-to-understand, syntactic specification of a programming language.

From certain classes of grammars, we can automatically build an efficient parser that determines the syntactic structure of a source program. As a side benefit, the parser construction process can reveal syntactic ambiguities and pain points that might have been overlooked during the initial phase of language design.

The structure imparted to a language by appropriately designed grammar is useful for translating source programs into correct object code, and for detecting errors.

A grammar allows a language to evolve or develop iteratively, adding new constructions to accomplish new tasks. These new constructions can be more easily integrated into an implementation that follows the grammatical structure of the language.

There are three general types of parsers for grammars: universal, descending, and ascending. Universal parsing methods such as the Cocke-YoungerKasami algorithm and Earley's algorithm can parse any grammar (see bibliographic notes). However, these general methods are too inefficient to be used in compiler production.

Methods that are commonly used in compilers can be classified as top-down or bottom-up. The more efficient top-down and bottom-up methods only work for subclasses of grammars, but several of these classes, especially the LL and LR grammars, are expressive enough to describe most of the syntactic constructions in modern programming languages.
Lexical errors include misspelling of identifiers, keywords, or operators; for example, the use of an identifier ellipce size instead of ellipse size, and the omission of quotation marks around the text that should be interpreted as a string.
Syntactic errors include incorrect placement of semicolons, plus extra or missing braces; that is, "{" or "}". As another example, in C or Java, the occurrence of a case statement without an enclosing switch statement is a syntactic error (however, this situation is generally accepted by the parser and caught later in processing, when the compiler tries to generate code).

Another reason to emphasize error recovery during parsing is that many errors appear to be syntactic, whatever their cause, and are exposed when parsing cannot continue. Some semantic errors, such as conflicts between types, can also be detected efficiently; however, accurately detecting semantic and logical errors at compile time is generally a difficult task.

• Report the presence of errors clearly and accurately.
• Recover from each error fast enough to be able to detect subsequent errors.
• Add minimal overhead to processing the correct programs.

Grammar Representation
Constructs that begin with keywords such as while or int are very easy to parse, as the keyword guides the choice of the grammar production to apply to match the input. Therefore, we will focus on expressions, which are challenging due to associativity and operator precedence. Associativity and precedence are resolved in the following grammar, which is similar to the ones we used in Chapter 2 to describe expressions, terms, and factors. E represents the expressions that consist of terms separated by the + signs, T represents the terms that consist of factors separated by the * signs, and F represents the factors that can be expressions between parentheses or identifiers:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

Handling of syntactic errors

The remainder of this section considers the nature of syntactic errors and general strategies for recovering from them. Two of these strategies, known as phrase-level and panic-mode recoveries, will be described in more detail in conjunction with specific parsing methods. If a compiler had to process only correct programs, its design and implementation would be greatly simplified. However, a compiler is expected to help the programmer locate and trace the bugs that inevitably creep into programs, despite the best efforts of the programmer. Incredibly, few languages are designed with error handling in mind, even though errors are so common. Our civilization would be radically different if spoken languages had the same requirements for syntactic precision as computer languages. Most programming language specifications do not describe how a compiler should respond to errors; their handling is the responsibility of the compiler designer. Planning to handle errors from the beginning can simplify the structure of a compiler and improve its ability to handle errors. Common programming errors can occur on many different levels.

- Lexical errors include misspelling of identifiers, keywords, or operators; for example, the use of an identifier ellipce size instead of ellipse size, and the omission of quotation marks around the text that should be interpreted as a string.
- Syntactic errors include incorrect placement of semicolons, plus extra or missing braces; that is, "{" or "}". As another example, in C or Java, the occurrence of a case statement without an enclosing switch statement is a syntactic error (however, this situation is generally accepted by the parser and caught later in processing, when the compiler tries to generate code).
- Semantic errors include type conflicts between operators and operands. An example is a return statement in a Java method, with the result type void.
- The logical errors can be anything from incorrect reasoning by the programmer in the use (in a C program) of the assignment operator =, instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the intent of the programmer.

The precision of parsing methods makes it possible to detect syntactic errors very efficiently. Various parsing methods, such as the LL and LR methods, detect an error as soon as possible; that is, when the token stream coming from the lexical analyzer cannot be further parsed according to the grammar for the language. More precisely, they have the property of viable prefix, which means that they detect the occurrence of an error as soon as they see a prefix of the input that cannot be completed to form a valid string in the language. Another reason to emphasize error recovery during parsing is that many errors appear to be syntactic, whatever their cause, and are exposed when parsing cannot continue. Some semantic errors, such as conflicts between types, can also be detected efficiently; however, accurately detecting semantic and logical errors at

compile time is generally a difficult task. The error handle in a parser has objectives that are simple to declare, but difficult to accomplish:

- Report the presence of errors clearly and accurately.
- Recover from each error fast enough to detect subsequent errors.
- Add minimal overhead to processing the correct programs.

Fortunately, common mistakes are simple, and a simple mechanism is often enough to handle them. How should an error handle report the presence of an error? At a minimum, you should report the place in the source program where an error was detected, since there is a good chance that the error itself occurred in one of the few previous tokens. A common strategy is to print the problem line with a pointer to the position where the error was detected.

Strategies for recovering from errors

Once an error is detected, how should the parser recover? Although there is no strategy that has been proven to be universally acceptable, some methods can be applied in many situations. The simplest method is for the parser to exit with an informational error message when it detects the first error. Additional errors are often discovered if the parser can restore itself to a state where it can continue processing the input, with reasonable hopes that further processing will provide useful diagnostic information. If errors pile up, it is better for the compiler to quit after exceeding a certain error limit than to produce an annoying flood of "false" errors. The remainder of this section is devoted to the following error recovery strategies: panic mode, phrase level, error productions, and global correction.

Panic mode recovery

With this method, when describing an error, the parser discards the input symbols, one at a time, until it finds a designated set of sync tokens. Synchronization tokens are generally delimiters such as semicolons or}, whose function in the source program is clear and unambiguous. The compiler designer must select the appropriate sync tokens for the source language. Although panic mode correction often skips a considerable amount of input without checking for additional errors, it has the advantage of being simple and, unlike certain methods we will consider later, it is guaranteed not to go into an infinite loop.

Phrase-level recovery

Upon discovering an error, a parser can perform a local correction on the remaining input; that is, you can substitute a prefix of the remaining entry for some string that allows you to continue. A common local fix is to replace a comma with a semicolon, remove a strange semicolon, or insert a missing semicolon. The choice of local

correction is left to the compiler designer. Of course, we must be careful to choose substitutions that do not take us into infinite cycles, as it would be, for example, if we always inserted something in the input ahead of the current input symbol. Phrase-level substitution has been used in various compilers that repair errors, as it can correct any input string. Its main disadvantage is the difficulty it has in coping with situations where the current error occurs before the point of detection.

Productions of errors

By anticipating the common errors that we might encounter, we can increase the grammar for the language, with productions that generate the erroneous constructions. A parser built from a grammar augmented by these error productions detects anticipated errors when an error output is used during parsing. Thus, the parser can generate appropriate error diagnostics on the erroneous construction that was recognized in the input.

Global correction

Ideally, a compiler should make the fewest changes in processing an incorrect input string. There are algorithms to choose a minimum sequence of changes, to obtain a correction with the lowest global cost. Given an incorrect input string x and a grammar G, these algorithms will look for a parse tree for a related string y, such that the number of insertions, deletions, and modifications of the tokens required to transform ax to y is the smallest possible. Unfortunately, these methods are generally too expensive to implement in terms of time and space, so these techniques are only of theoretical interest at this time. Note that a nearly correct program may not be what the programmer had in mind. However, the notion of least-cost correction provides a standard for evaluating error recovery techniques, which has been used to find optimal replacement strings for phrase-level recovery.

Context-free grammars

If we use a syntactic variable instr to denote instructions, and a variable expr to denote expressions, the following output:

$$instr \rightarrow \textbf{if} \ (\ expr\ )\ instr\ \textbf{else}\ instr$$

specifies the structure of this form of conditional statement. So other productions precisely define what an expr is and what else an instr can be. In this section we will review the definition of a context-free grammar and introduce the terminology to talk about parsing. In particular, the notion of derivations is very useful for discussing the order in which productions are applied during parsing.

The formal definition of a context-free grammar

1. Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal"; we will often use the word "token" instead of terminal, when it is clear that we are talking only about the name of the token. We assume that the terminals are the first components of the tokens that the lexical analyzer produces. The terminals are the if and else reserved words, and the "(" and ")" symbols.

2. Nonterminals are syntactic variables that denote sets of strings. Instr and expr are nonterminal. The sets of strings denoted by the nonterminals help define the language generated by the grammar. Non-terminals impose a hierarchical structure on the language, which represents the key to parsing and translation.

3. In a grammar, a nonterminal is distinguished as the initial symbol, and the set of strings it denotes is the language generated by the grammar. By convention, the productions for the initial symbol are listed first.

4. The productions of a grammar specify how the terminals and non-terminals can be combined to form strings. Each production consists of: (a) A non-terminal, known as the header or left side of the production; this production defines some of the strings denoted by the header. (b) The symbol →. Sometimes :: = has been used instead of the arrow. (c) A body or right side, consisting of zero or more terminals and non-terminals. The body components describe a way that nonterminal strings can be constructed in the header.

Notation Conventions

To avoid always having to say that "these are the terminals," "these are the non-terminals," and so on, we will use the following notation conventions for grammars throughout the rest of this book:

1. These symbols are terminals:

   (a) The first lowercase letters of the alphabet, such as a, b, c.

   (b) Operator symbols like +, *, and so on.

   (c) Punctuation symbols such as parentheses, commas, and so on.

   (d) The digits 0, 1,…, 9. (e) Strings in bold such as id or if, each of which represents a single terminal symbol.

2. These symbols are non-terminal:

   (a) The first capital letters of the alphabet, such as A, B, C.

(b) The letter S, which, when it appears, is usually the initial symbol.

(c) Italicized and lowercase names, such as expr or instr.

(d) When talking about programming constructs, capital letters can be used to represent non-terminals. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

3. The last capital letters of the alphabet, such as X, Y, Z, represent grammatical symbols; that is, they can be non-terminal or terminal.
4. The last lowercase letters of the alphabet, such as u, v,..., z, represent strings of terminals (possibly empty).
5. The lowercase Greek letters α, β, γ, for example, represent strings (possibly empty) of grammatical symbols. Hence, a generic output can be written as A → α, where A is the header and α the body.
6. A set of productions A → α1, A → α2,..., A → αk with a common header A (we will call them productions A), can be written as A → α1 | α2 | ... | αk. We call α1, α2,..., αk the alternatives for A.
7. Unless otherwise indicated, the heading of the first production is the initial symbol.

Derivations

The construction of a parse tree can be made precise if we take a derivational view, in which productions are treated as rewriting rules. Starting with the initial symbol, each rewrite step replaces a nonterminal with the body of one of its productions. This derivational view corresponds to the top-down construction of a parsing tree, but the precision provided by the derivations will be very useful when we talk about bottom-up parsing. As we will see, bottom-up parsing relates to a class of taps known as “right-most” taps, where the right-hand nonterminal is rewritten at each step.

For example, consider the following grammar, with a single nonterminal E, which adds a production E → - E to the grammar:

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathrm{id}$$

The production E → - E means that if E denotes an expression, then - E must also denote an expression . The substitution of a single E for -E will be described by writing the following:

$$E \Rightarrow -E$$

which is read as "E derives -E". The production E → (E) can be applied to substitute any instance of E in any string of grammatical symbols for (E); for example, E ∗ E ⇒ (E) ∗ E or E ∗ E ⇒ E ∗ (E). We can take a single E and apply productions repeatedly and in any order to obtain a sequence of substitutions. For example,

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

we call such a sequence of substitutions a derivation of - (id) from E. This derivation provides proof that the string - (id) is a specific instance of an expression. For a general definition of derivation, consider a nonterminal A in the middle of a sequence of grammatical symbols, as in αAβ, where α and β are arbitrary strings of grammatical symbols. Suppose A → γ is a production. So, we write αAβ ⇒ αγβ. The symbol ⇒ means, “is derived in one step”. When a sequence of derivation steps α1 ⇒ α2 ⇒… ⇒ αn is rewritten as α1 to αn, we say that α1 derives αn. It is often convenient to be able to say, "drift in zero or more steps." For this purpose, we can use the symbol ⇒ ∗. Thus,

1. $\alpha \overset{*}{\Rightarrow} \alpha$, para cualquier cadena α.

2. Si $\alpha \overset{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$, entonces $\alpha \overset{*}{\Rightarrow} \gamma$.

Similarly, ⇒ means "drift in one or more steps". If S ⇒ ∗ α, where S is the initial symbol of a G grammar, we say that α is a phrase form of G. Notice that a phrase form can contain both terminals and non-terminals, and can be empty. A statement of G is a phrase form without nonterminal symbols. The language generated by a grammar is its set of sentences. Hence, a string of terminals w is in L (G), the language generated by G, if and only if w is a statement of G (or S ⇒ ∗ w). A language that can be generated by a grammar is considered a context-free language. If two grammars generate the same language, they are considered equivalent.

Each nonterminal is replaced by the same body in the two derivations, but the order of the substitutions is different. To understand how parsers work, we must consider the derivations in which the nonterminal to be substituted in each step is chosen as follows:

1. In derivations from the left, the no is always chosen. terminal on the left in each sentence. If α ⇒ β is a step in which the nonterminal is replaced by the left in α, we write α ⇒ lm β.

2. In right-hand derivations, the right nonterminal is always chosen; in this case we write α ⇒rm β.

The derivation (4.8) is from the left, so it can be rewritten as follows:

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+E) \underset{lm}{\Rightarrow} -(\mathbf{id}+\mathbf{id})$$

Using our notation conventions, each step to the left can be written as wAγ ⇒ lm wδγ, where w consists only of terminals, A → δ is the output that is applied, and γ is a string of grammar symbols. To emphasize that α derives β by a derivation from the left, we write α ⇒ ∗ lm β. If S ⇒ ∗ lm α, we say that α is a left sentence form of the grammar in question. The same definitions are valid for leads from the right. These derivations are sometimes known as canonical derivations.

Parse trees and derivations

A parse tree is a graphical representation of a derivation that filters the order in which productions are applied to replace nonterminals. Each interior node in a parse tree represents a production application. The inner node is labeled with the nonterminal A in the output header; the children of the node are labeled, left to right, by the symbols in the body of the production this A was substituted for during derivation.

The leaves of a parse tree are labeled by nonterminals or terminals and, read from left to right, constitute a phrase form, which is called the product or boundary of the tree. To see the relationship between derivations and parse trees, consider any derivation α1 ⇒ α2 ⇒… ⇒ αn, where α1 is a single nonterminal A. For each sentence form αi in the derivation, we can construct a tree of syntactic analysis whose product is αi. The process is an induction on i.

BASE: The tree for α1 = A is a single node, labeled A.

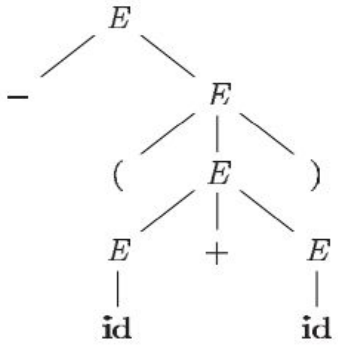


Figura 4.3: Árbol de análisis sintáctico para $-(\mathbf{id} + \mathbf{id})$

INDUCTION: Suppose we have already built a parse tree with the product αi − 1 = X1X2… Xk (note that according to our conventions of notation, each grammatical symbol Xi is a nonterminal or a terminal). Suppose αi is derived from αi − 1 by substituting Xj, a nonterminal, for β = Y1Y2… Ym. That is, in the i-th step of the derivation, the production Xj → β is applied to αi − 1 to derive αi = X1X2… Xj − 1βXj + 1… Xk. To model this step in the derivation, we search for the jth leaf, starting from the left, in the current parse tree. This sheet is labeled Xj. We give this sheet m children, labeled Y1, Y2,…, Ym, starting from the left. As a special case, if m = 0 then β =, and we provide the jth leaf with a child labeled as.

Ambiguity

For most parsers, it is desirable that the grammar be unambiguous, otherwise we cannot uniquely determine which parse tree to select for a statement. In other cases, it is convenient to use carefully chosen ambiguous grammars, along with disambiguation rules, which "discard" unwanted syntactic trees, leaving only one tree for each statement.

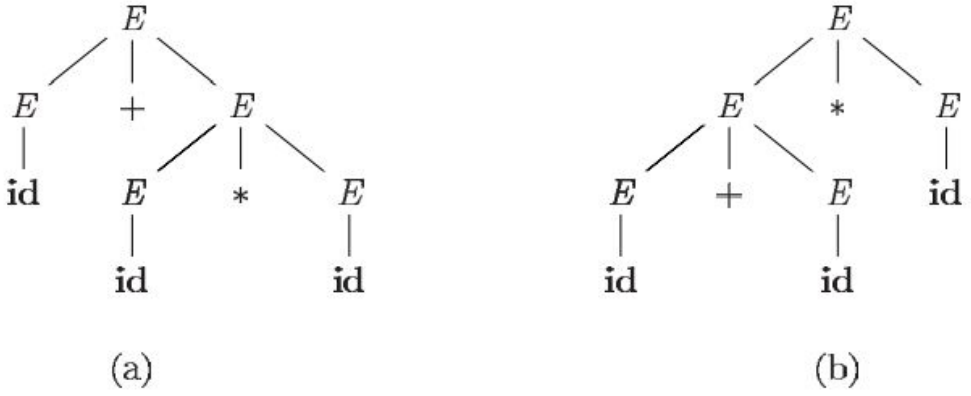


Figura 4.5: Dos árboles de análisis sintáctico para $\mathbf{id}+\mathbf{id}*\mathbf{id}$

Verifying the Language Generated by a Grammar

Although compiler designers rarely do this for a complete programming language grammar, it is useful to be able to reason that a given set of productions generates a specific language. Problematic constructions can be studied by writing a concise, abstract grammar and

studying the language it generates. Next we are going to build a grammar of this type, for conditional instructions. A proof that a grammar G generates a language L consists of two parts: showing that all strings generated by G are in L and, conversely, that all strings in L can certainly be generated by G.

Comparison between free grammars Context and Regular Expressions

Before leaving this section on grammars and their properties, let's state that grammars are a more powerful notation than regular expressions. Every construction that can be described by a regular expression can be described by a grammar, but not the other way around. Alternatively, each regular language is a context-free language, but not the other way around. For example, the regular expression (a | b) ∗ abb and the following grammar:

$$
\begin{aligned}
A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\
A_1 &\rightarrow bA_2 \\
A_2 &\rightarrow bA_3 \\
A_3 &\rightarrow \epsilon
\end{aligned}
$$

describe the same language, the set of strings of as and bs ending in abb. We can mechanically construct a grammar to recognize the same language as a non-deterministic finite automaton (NFA). The previous grammar was constructed from the AFN of figure 3.24, using the following construction:

1. For each state i of the AFN, create a non-terminal Ai.
2. If state i has a transition to state j with input a, add the output Ai → aAj. If state i goes to state j with the input, add the production Ai → Aj.
3. If i is an accepting state, add Ai →.
4. If i is the initial state, make Ai the initial symbol of the grammar.

Writing a grammar

Grammars are capable of describing almost most of the syntax of programming languages. For example, the requirement that identifiers must be declared before use cannot be described by a context-free grammar. Therefore, the sequences of tokens that a parser accepts form a superset of the programming language; subsequent stages of the compiler should parse the output of the parser, to ensure that it complies with the rules that the parser does not check.

Comparison between lexical analysis and parsing it

would be reasonable to ask: "Why use regular expressions to define the lexical syntax of a language?" There are several reasons.

1. By separating the syntactic structure of a language into lexical and non-lexical parts, a convenient way is provided to module the user interface of a compiler into two components of a manageable size.

2. The lexical rules of a language are often quite simple, and to describe them we do not need a notation as powerful as grammars.

3. In general, regular expressions provide a more concise and easy-to-understand notation for tokens compared to grammars.

4. More efficient lexical analyzers can be built automatically from regular expressions, compared to arbitrary grammars.

There are no firm guidelines on what to put in the lexical rules, as opposed to the syntactic rules. Regular expressions are very useful for describing the structure of constructs such as identifiers, constants, reserved words, and whitespace. On the other hand, grammars are very useful for describing nested structures, such as balanced parentheses, related begin-end statements, corresponding if-then-else statements, and so on. These nested structures cannot be described by regular expressions.

Elimination of ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate ambiguity. As an example, let's disambiguate the following "hanging else" grammar:

$$
\begin{array}{lcl}
\mathit{instr} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{instr} \\
& | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{instr} \textbf{ else } \mathit{instr} \\
& | & \textbf{otra}
\end{array}
$$

Here, "other" represents any other statement. According to this grammar, the following compound conditional statement:

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

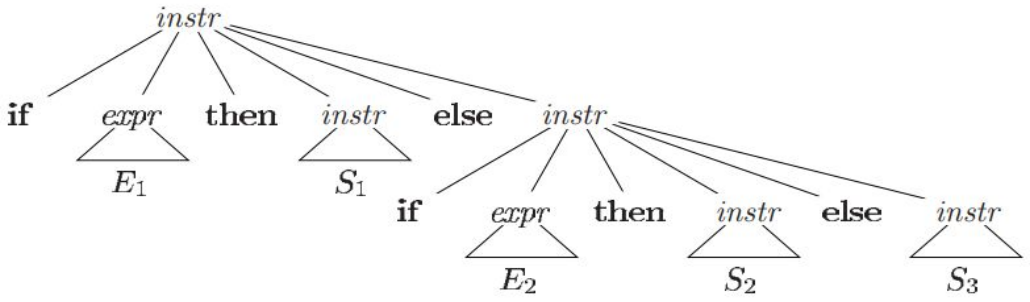


Figura 4.8: Árbol de análisis sintáctico para una instrucción condicional

Elimination of left recursion

A grammar is left recursive if it has a terminal A such that there is a derivation A ⇒ Aα for a certain string α. Top-down parsing methods cannot handle left recursive grammars, so a transform is needed to remove left recursion.

The immediate left recursion can be eliminated by the following technique, which works for any number of A productions. First, the productions are grouped as follows:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

where no βi ends with an A. Then the productions are substituted A using the following:

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

The nonterminal A generates the same strings as before, but is no longer left recursive. This procedure removes all left recursion from productions A and A (as long as no αi is), but does not remove left recursion that includes derivations of two or more steps. For example, consider the following grammar:

$$S \rightarrow A\ a \mid b$$
$$A \rightarrow A\ c \mid S\ d \mid \epsilon$$

The nonterminal S is left recursive, since S ⇒ Aa ⇒ Sda, but is not immediately left recursive.

Cabrera Ramírez Gerardo

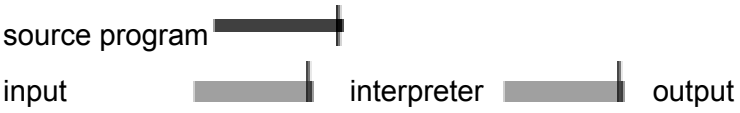
Compiler: it is a program that can read programs in one language (the source language) and translate it into a program in another language (thelanguage target).

source program compiler target program

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

input program destination output

Interpreter: This instead of producing a destination program as a translation, it gives us the appearance of directly executing the operations specified in the source program (source) with the inputs provided by the user.



In the structure of a compiler there are two processes:

Analysis: That divides the source program into components and imposes a grammatical structure on them. If this part detects a syntax error in the source program, the source program should send an error message to the user to correct it.

Synthesis: Build the desired destination program from the representation intermediate and the information in the symbol table.

Lexical analysis: This reads the flow of characters that make up the source program and groups them into meaningful sequences, known as lexemes. For each lexeme, the lexical analyzer outputs a token of the form: token-name, attribute-value

Syntactic analysis: Uses the first components of the tokens generated by the lexical analyzer to create a representation that describes the grammatical structure of the flow of tokens.

Semantic analysis: Uses the syntactic tree and the information in the symbol table to check the semantic consistency of the source program with the language definition. The important thing about this is type checking (verification), where the compiler checks that each operator has matching operands.

Code generation: 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 are selected for each variable that the program uses.

And as we all know, machine language is the one that computers can read like all its components, be it RAM, processor, etc. and the high level is the one that the user can order the machine to do through instructions in our language.

Syntactic Analysis (part 1)

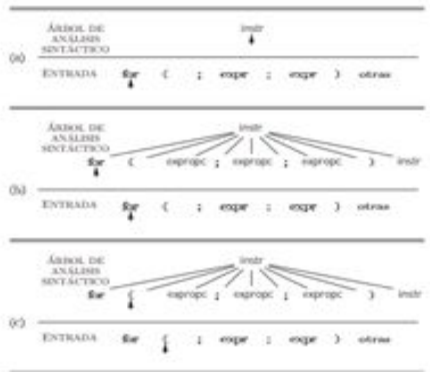
It is the process of determining how a chain of terminals can be generated using a grammar.
This section introduces a parsing method known as "recursive descent", this can be used both for parsing and for the implementation of syntax-oriented translators.

Most parsing methods conform to one of two classes, called top-down and bottom-up methods. These refer to the order in which nodes are built in the parse tree. In descending type analyzers, the construction begins at the root and proceeds towards the leaves, while in ascending type analyzers, the construction begins in the leaves and proceeds towards the root.

Top-down parsing The top-down
construction of a parse tree like the one in figure 2.17 is done starting with the root, labeled with the initial nonterminal instr

instr → expr;
| if (expr) instr
| for (expropc; expropc; expropc) instr
| other

expropc → ϵ
| expr



Predictive parsing
Here we will consider a simple form of recursive descent parsing, known as predictive parsing, in which the pre-parse symbol unambiguously determines the flow of control through the procedure body for each nonterminal.

Syntactic trees for instructions

Figura 2.39: Construcción de árboles sintácticos para expresiones e instrucciones

Syntactic trees for expressions

SINTAXIS CONCRETA	SINTAXIS ABSTRACTA
=	asigna
||	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
$-_{unario}$	menos
[]	acceso

Lexical Analysis (parts 2 and 3)

A lexical analyzer reads the characters from the input and groups them into “token objects”. Along with a terminal token that is used for parsing decisions, a token object carries additional information in the form of attribute values.

The lexical analyzer in this section allows numbers, identifiers, and “whitespace” (spaces, tabs, and newlines) to appear within expressions.

Read Ahead

If the next character is =, then> is part of the character sequence> =, the lexeme for the operator token "greater than or equal to". Otherwise,> by itself forms the "greater than" operator, and the lexical analyzer has read an extra character.

A general method of reading ahead of input is to maintain an input buffer, from which the lexical analyzer can read and return characters.

A lexical analyzer
So far in this section, the pseudocode snippets are put together to form a function called scan that returns token objects, as follows:

```
Token scan () {
 omit whitespace;
handle the numbers;
manage reserved words and identifiers;
/ * If we get here, treat the look-ahead read character as token * / Token t = new Token (look);
look = blank / * initialization, as we saw before * /;
return t;
 }
```

Tokens, patterns, and lexemes

- A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol that represents a type of lexical unit; for example, a specific keyword or a sequence of input characters denoting an identifier. The token names are the input symbols that are processed by the parser. From now on, we will generally write the name of a token in bold. We will often refer to a token by name.
- A pattern is a description of the form that the lexemes of a token can take. In the case of a keyword like token, the pattern is just the sequence of characters that make up the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is related through many strings.
- A lexeme is a sequence of characters in the source program, matching the pattern for a token, and identified by the lexical analyzer as an instance of that token.

TOKEN	DESCRIPCIÓN INFORMAL	LEXEMAS DE EJEMPLO
if	caracteres `i`, `f`	`if`
else	caracteres `e`, `l`, `s`, `e`	`Else`
comparacion	`<` o `>` o `<=` o `>=` o `==` o `!=`	`<=`, `!=`
id	letra seguida por letras y dígitos	`pi`, `puntuacion`, `D2`
numero	cualquier constante numérica	`3.14159`, `0`, `6.02e23`
literal	cualquier cosa excepto ", rodeada por "'s	`"core dumped"`

Attributes for tokens

When more than one lexeme can match a pattern, the parser must provide subsequent compiler stages with additional information about the specific lexeme that matched.

Other functions it performs:
• Delete comments from the program.
• Eliminate blank spaces, tabs, carriage return, etc., and in general, everything that lacks meaning according to the syntax of the language.
• Recognize user identifiers, numbers, reserved words of the language, ..., and treat them correctly with respect to the symbol table (only in cases where you should deal with the symbol table).
• Keep track of the line number you are reading from, in case an error occurs, give information about where it occurred.

Example: The token names and associated attribute values for the following Fortran statement:
E = M * C ** 2
are then written as a sequence of pairs.
<id, pointer to the entry in the symbol table for E>
<assign–op>
<id, pointer to the entry in the symbol table for M>
<mult–op>
<id, pointer to the entry in the table of symbols for C>
<exp–op>
<number, integer value 2>

Note that in certain pairs especially in operators, punctuation marks and keywords, there is no need for an attribute value.

Lexical Errors
Without the help of the other components it is difficult for a Lexical Analyzer to know that there is an error in the source code.

Example:
fi (a == f (x)) ...

A lexical analyzer cannot tell if fi is a misspelled if keyword, or a handle to an undeclared function. Since fi is a valid lexeme for the id token, the parser should return the id token to the parser and let some other phase of the compiler (perhaps the parser in this case) send an error due to the transposition of the letters.

The simplest recovery strategy is "panic mode" recovery. We remove successive characters from the rest of the input, until the lexical analyzer can find a well-formed

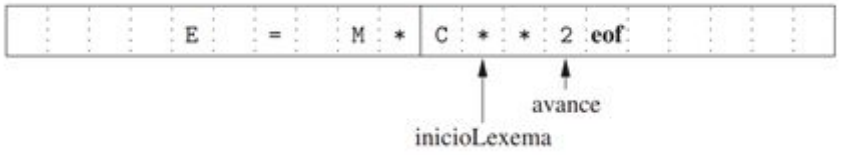
token at the beginning of what is left of the input. This retrieval technique can confuse the parser, but in an interactive computing environment, it can be quite suitable.

Other possible error recovery actions are:
1. Remove a character from the rest of the input.
2. Insert a missing character in the rest of the entry.
3. Replace one character with another.
4. Transpose two adjacent characters.

Buffering input
This task is made difficult because we often have to look for one or more characters beyond the next lexeme in order to be sure that we have the correct lexeme.

Buffer Pairs
Specialized buffer usage techniques have been developed to reduce the amount of overhead required in processing a single input character.

One important scheme involves the use of two buffers that are reloaded alternately:



Each buffer is the same size N, and typically N is the size of a disk block (that is, 4 096 bytes). By using a system read command we can read N characters and put them in a buffer, instead of using a system call for each character.

Two pointers are kept to the input:
1. Thepointer startLexema marks the beginning of the current lexeme, whose extension we are trying to determine.
2. The forward pointer scans ahead until it finds a match in the pattern; During the remainder of the chapter we will cover the exact strategy by which this determination is made.

Specifying tokens
Regular expressions are an important notation for specifying lexeme patterns. Although they cannot express all possible patterns, they are very effective in specifying the types of patterns that we actually need for tokens.

Operations in languages Language
concatenation is when all the strings that are formed by taking a string from the first language and a string from the second language are concatenated, in all possible ways. The lock (Kleene) of a language L, denoted as L *, is the set of strings that are obtained by concatenating L zero or more times. Note that L0, the "concatenation of L zero times," is defined as {}, and by induction, Li is Li − 1 L. Finally, the positive lock, denoted L +, is the same as the Kleene lock, but without the L0 term. That is, it will not be in L + unless it is in the same L.

OPERACIÓN	DEFINICIÓN Y NOTACIÓN
Unión de L y M	$L \cup M = \{ s \mid s \text{ está en } L \text{ o } s \text{ está en } M \}$
Concatenación de L y M	$LM = \{st \mid s \text{ está en } L \text{ y } t \text{ está en } M \}$
Cerradura de Kleene de L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Cerradura positivo de L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Regular definitions
If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:
d1 → r1
d2 → r2
...
dn → rn
Where:
1. Each di is a new symbol, which is not in Σ and is not the same as any other d. 2. Each ri is a regular expression over the alphabet Σ ∪ {d1, d2,..., di − 1}.
By restricting ri to Σ and to the previously defined ds, we avoid recursive definitions and can build a regular expression on only Σ, for each ri.

Token recognition
Now we need to study how to take all the patterns for all the necessary tokens and build a piece of code to examine the input string and look for a prefix that is a lexeme that matches one of those patterns.

instr → if expr then instr
| if expr then instr else instr
| ϵ
expr → term opre term
| term
term → id
| number

For oprel, we use the comparison operators of languages such as Pascal or SQL, where = is "is equal to" and <> is "is not equal to", since it presents an interesting structure of lexemes.
The terminals of the grammar, which are if, then, else, oprel, id and number, are the names of tokens as far as the lexical analyzer is concerned.

LEXEMAS	NOMBRE DEL TOKEN	VALOR DEL ATRIBUTO
Cualquier *ws*	–	–
`if`	**if**	–
`Then`	**then**	–
`else`	**else**	–
Cualquier *id*	**id**	Apuntador a una entrada en la tabla
Cualquier *numero*	**numero**	Apuntador a una entrada en la tabla
`<`	**oprel**	`LT`
`<=`	**oprel**	`LE`
`=`	**oprel**	`EQ`
`<>`	**oprel**	`NE`
`>`	**oprel**	`GT`
`>=`	**oprel**	`GE`

State transition diagrams
In the construction of a lexical analyzer, we first convert the patterns into stylized flow diagrams, which are called "state transition diagrams".
State transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that might occur during the process of exploring the input, looking for a lexeme that matches one of several patterns.
The lines are directed from one state to another on the state transition diagram. Each line is labeled by a symbol or set of symbols. If we are in a certain state s, and the next input symbol is a, we look for a line that comes out of state s and is labeled by a (and perhaps other symbols as well).

Example:

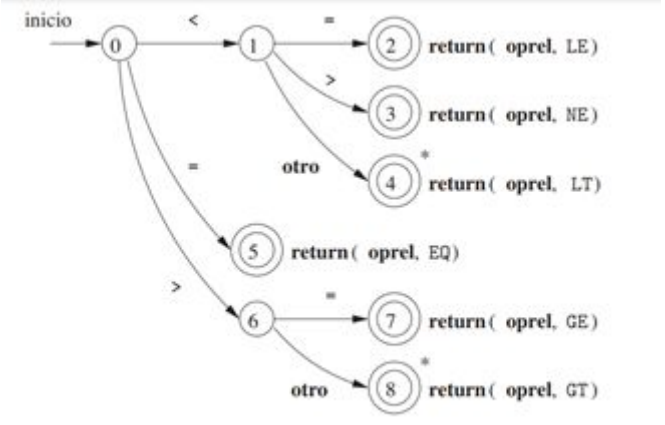


Figura 3.13: Diagrama de transición de estados para **oprel**

Recognition of reserved words and identifiers

Recognition of reserved words and identifiers presents a problem. Keywords like if or then are generally reserved (as in our sketch), so they are not identifiers, even though they appear to be.

To find identifier lexemes, this diagram will also recognize the if, then, and else keywords from our sketch.

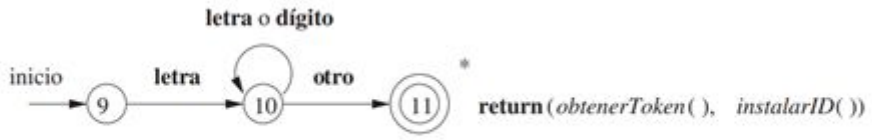


Figura 3.14: Un diagrama de transición de estados para identificadores (**id**) y palabras clave

Completing the sketch

The transition diagram for the id tokens that we saw in the figure has a simple structure. Starting at state 9, check that the lexeme begins with a letter and goes to state 10 if it does. We remain in state 10 as long as the input contains letters and digits. When we find the first character that is not a letter or digit, we go to state 11 and accept the lexeme found.

Architecture of a lexical parser based on state transition diagrams

There are several ways in which state transition diagrams can be used to build a lexical parser. Each state is represented by a piece of code. The code for a state is itself a switch statement or a multi-way branch that determines the next state through the process of reading and examining the next input character.

Example:

In figure 3.18 we see a sketch of getOpRel (), a function in C ++ whose job is to simulate the transition diagram of figure 3.13 and return an object of type TOKEN

```
TOKEN obtenerOpRel()
{
    TOKEN tokenRet = new (OPREL);
    while(1) { /* repite el procesamiento de caracteres hasta que
                  ocurre un retorno o un fallo */
        switch(estado) {
            case 0: c = sigCar();
                    if ( c == '<' ) estado = 1;
                    else if ( c == '=' ) estado = 5;
                    else if ( c == '>' ) estado = 6;
                    else fallo(); /* el lexema no es un oprel */
                    break;
            case 1: ...
            ...
            case 8: retractar();
                    tokenRet.atributo = GT;
                    return(tokenRet);
        }
    }
}
```

Figura 3.18: Bosquejo de la implementación del diagrama de transición **oprel**

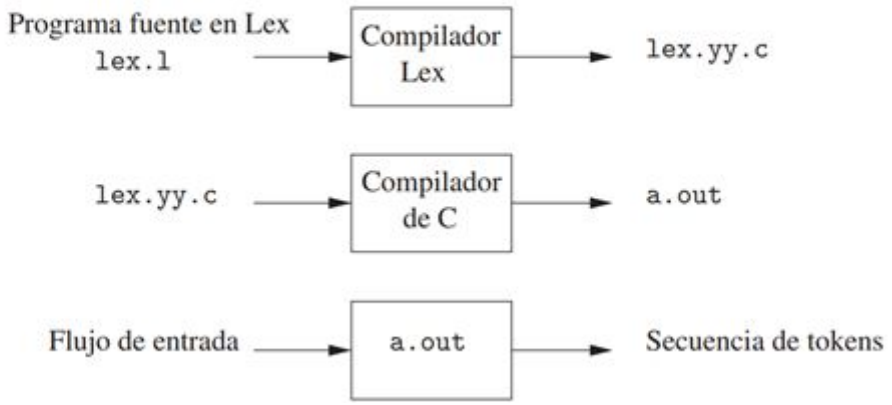
The Lex lexical parser generator

The input notation for the tool Lex is known as the Lex language, and the tool itself is the Lex compiler. The Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram.

Using Lex

An input file, which we will call lex.1, is written in the Lex language and describes the lexical analyzer to be generated. The Lex compiler transforms Lex.1 into a C program, into a file that is always called lex.yy.c. The C compiler compiles this file into a file called a.out, as usual. The output of the C compiler is a functional lexical analyzer, which can receive a stream of input characters and produce a string of tokens.

El generador de analizadores léxicos Lex



Structure of LexLex

programs Aprogram has the following form:

declarations
%%
translation rules
%%
helper functions

Each pattern is a regular expression, which can use the regular definitions from the declarations section. Actions are snippets of code, typically written in C, although many variants of Lex have been created that use other languages.
The lexical parser that Lex creates works in conjunction with the parser as follows. When the parser calls the parser, it starts reading the rest of your input, one character at a time, until it finds the longest prefix of the input that matches one of the Pi patterns. Then it executes the associated action Ai. Usually Ai will return to the parser, but if it doesn't (perhaps because Pi describes whitespace or comments), then the parser proceeds to look for additional lexemes, until one of the corresponding actions causes a return. to the parser. The parser returns a single value, the name of the token, to the parser, but uses the shared integer variable yylval to pass additional information about the found lexeme to it, if necessary.

Conflict resolution in Lex
We have referred to the two rules that Lex uses to decide on the appropriate lexeme to select, when several prefixes of the input match one or more patterns:
1. Always prefer a longer prefix to a shorter one.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Finite automata
These consist of graphs like state transition diagrams, with some differences:
1. Finite automata are recognizers; they just say "yes" or "no" in relation to each possible input string.
2. Finite automata can be of two types:
(a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their lines. A symbol can label multiple lines arising from the same state, and the empty string is a possible label.
(b) Deterministic finite automata (AFD) have, for each state, and for each symbol of their input alphabet, exactly one line with that symbol coming out of that state.

Non-deterministicnon-deterministic
finite automata Afinite automata (NFA) consists of:
1. A finite set of states S.

2. A set of input symbols Σ, the input alphabet. We assume that, representing the empty string, it will never be a member of Σ.
3. A transition function that provides, for each state and for each symbol in Σ ∪ {}, a set of subsequent states.
4. A state s0 from S, distinguished as the initial state.
5. A set of states F, a subset of S, which are distinguished as the accepting states (or final states).

Transition tables
We can also represent an AFN by means of a transition table, whose rows correspond to the states, and whose columns correspond to the input symbols already. The input for a given state and the input is the value of the transition function that applies to those arguments. If the transition function has no information about that state-input pair, we place ∅ in the table for that state.
Example:

Autómatas finitos

ESTADO	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Deterministic finite
automata A deterministic finite automata (AFD) is a special case of an AFN, where:

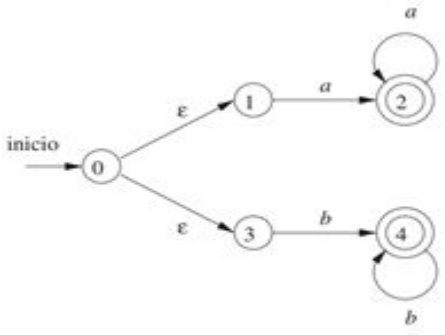


Figura 3.26: AFN que acepta a **aa*|bb***

1. There are no movements at the input.
2. For each state s and each input symbol a, there is exactly one line arising from s, Undecidable as a.
If we use a transition table to represent an AFD, then each input is a single state. Therefore, we can represent this state without the keys that we use to form the sets.

While the AFN is an abstract representation of an algorithm to recognize the strings of a certain language, the AFD is a simple and concrete algorithm to recognize strings.
Example AFN:

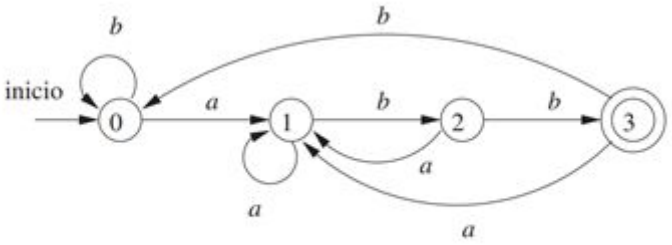


Figura 3.28: AFD que acepta a (a|b)*abb

The structure of the generated parser
Figure 3.49 presents the generalities about the architecture of a lexical parser generated by Lex. The program that serves as a lexical analyzer includes a fixed program that simulates an automaton; at this point we leave open the decision of whether the automaton is deterministic or not. The rest of the lexical analyzer consists of components that are created from the Lex program, by Lex himself.

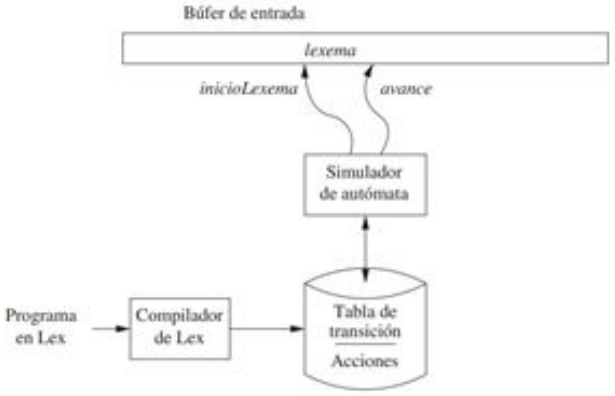


Figura 3.49: Un programa en Lex se convierte en una tabla de transición y en acciones, para que las utilice un simulador de autómatas finitos

These components are:
1. A transition table for the automaton.
2. The functions that are passed directly through Lex on exit.
3. The actions of the input program, which appear as code snippets that the automaton simulator must invoke at the appropriate time.

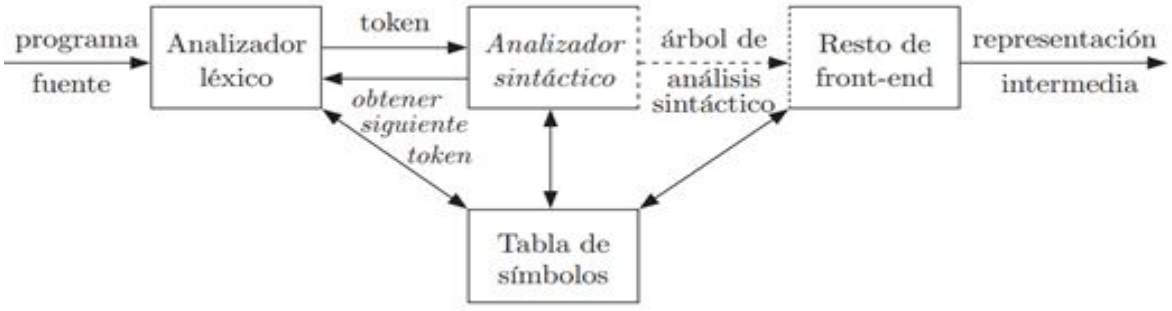
AFD-based pattern matching search engine optimization
1. The first algorithm is useful in a Lex compiler, as it builds an AFD directly from a regular expression, without building an intermediate AFN. Also, the resulting AFD can have fewer states than the AFD that is built using an AFN.

2. The second algorithm minimizes the number of states of any AFD, by combining the states that have the same future behavior. The algorithm itself is quite efficient, since it is executed in a time O(n log n), where n is the number of states of the AFD.
3. The third algorithm produces more compact representations of the transition tables than the standard two-dimensional table.

Parsing

The function of the parser

the parser obtains a string of tokens from the lexical analyzer, as shown in the image:



Verifies and the name string tokens cangenerated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible form and to recover from frequently occurring errors to continue processing the rest of the program. For well-formed programs, the parser builds a parsing tree and passes it to the rest of the compiler for further processing.

There are three general types of parsers for grammars: universal, descending, and ascending.
Universal parsing methods like the Cocke-Younger Kasami algorithm and the Earley algorithm can parse any grammar. However, these general methods are too inefficient to be used in compiler production.

Methods that are commonly used in compilers can be classified as top-down or bottom-up.
Top-down methods: Build parsing trees from top (root) to bottom (leaves).
Ascending methods: They start from the leaves and work their way to the root.
In either case, the input to the parser is scanned from left to right, one symbol at a time.

Grammar Representation

Constructs that begin with keywords such as while or int are very easy to parse, as the keyword guides the choice of the grammar output to be applied to match the input. Associativity and precedence are resolved in the following grammar:

To describe expressions, terms, and factors. E represents expressions that consist of terms separated by + signs, T represents terms that consist of factors separated by * signs, and F represents factors that can be expressions between parentheses or identifiers:

E → E + T | T
T → T ∗ F | F (4.1)
F → (E) | id
The grammar for expressions (4.1) belongs to the class of LR grammars that are suitable for upstream parsing.

Handling Syntactic
Errors Common programming errors can occur at many different levels.

- Lexical errors include misspelling of identifiers, keywords, or operators; for example, the use of an identifier ellipce size instead of ellipse size, and the omission of quotation marks around the text that should be interpreted as a string.
- Syntactic errors include incorrect placement of semicolons, plus extra or missing braces; that is, "{" or "}". As another example, in C or Java, the occurrence of a case statement without an enclosing switch statement is a syntactic error (however, this situation is generally accepted by the parser and caught later in processing, when the compiler tries to generate code).
- Semantic errors include type conflicts between operators and operands. An example is a return statement in a Java method, with the result type void.
- The logical errors can be anything from incorrect reasoning by the programmer in the use (in a C program) of the assignment operator =, instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the intent of the programmer.

The error handle in a parser has objectives that are simple to declare, but difficult to accomplish:

- Report the presence of errors clearly and accurately.
- Recover from each error fast enough to detect subsequent errors.
- Add minimal overhead to processing the correct programs.

Strategies for recovering from errors

Panic mode recovery
With this method, when describing an error the parser discards the input symbols, one at a time, until it finds a designated set of sync tokens

Although the panic mode fix to Often skipping a considerable amount of input without checking for additional errors, it has the advantage of being simple and unlike certain methods.

Phrase-level recovery

Upon discovering an error, a parser can perform a local correction on the remaining input; that is, you can substitute a prefix of the remaining entry for some string that allows you to continue. The choice of local correction is left to the compiler designer. Of course we must be careful to choose substitutions that do not take us into infinite cycles.

Phrase-level substitution has been used in various compilers that repair errors, as it can correct any input string.

Productions of errors

By anticipating the common errors that we might encounter, we can increase the grammar for the language, with productions that generate erroneous constructions.

A parser built from a grammar augmented by these error productions detects anticipated errors when an error output is used during parsing. Thus, the parser can generate appropriate error diagnostics on the erroneous construction that was recognized in the input.

Global correction

Ideally, a compiler should make the fewest changes in processing an incorrect input string.

Given an incorrect input string x and a grammar G, these algorithms will look for a parse tree for a related string y, such that the number of insertions, deletions, and modifications of the tokens required to transform ax to y is the smallest possible.

Context-free grammars

If we use a syntactic variable instr to denote instructions, and a variable expr to denote expressions, the following output:

instr → if (expr) instr else instr (4.4)

specifies the structure of this form of conditional statement. So other productions precisely define what an expr is and what else an instr can be.

The formal definition of a context-free grammar

1. Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal"; we will often use the word "token" instead of terminal, when it is clear that we are talking only about the name of the token. We assume that the terminals are the first components of the tokens that the lexical

analyzer produces. In (4.4), the terminals are the if and else keywords, and the symbols "(" and ")".

2. Nonterminals are syntactic variables that denote sets of strings. In (4.4), instr and expr are nonterminals. The sets of strings denoted by the nonterminals help define the language generated by the grammar. Non-terminals impose a hierarchical structure on the language, which represents the key to parsing and translation.
3. In a grammar, a nonterminal is distinguished as the initial symbol, and the set of strings it denotes is the language generated by the grammar. By convention, the productions for the initial symbol are listed first.
4. The productions of a grammar specify how the terminals and non-terminals can be combined to form strings. Each production consists of:
   (a) - A non-terminal, known as the header or left side of the production; this production defines some of the strings denoted by the header.
   (b) - The symbol →. Sometimes :: = has been used instead of the arrow.
   (c) A body or right side, consisting of zero or more terminals and non-terminals. The body components describe a way that nonterminal strings can be constructed in the header.

In this grammar, the terminal symbols are:
id + - * / ()

The nonterminal symbols are expression, term, and factor, and expression is the initial symbol.

expression → expression + term
expression → expression - term
expression → term
term → term * factor
term → term / factor
term → factor
factor → (expression)
factor → id

Notation Conventions

1. These symbols are terminals:
(a) The first letters lowercase letters of the alphabet, such as a, b, c.
(b) Operator symbols like +, *, and so on.
(c) Punctuation symbols such as parentheses, commas, and so on.
(d) The digits 0, 1,..., 9.
(e) Strings in bold such as id or if, each of which represents a single terminal symbol.

2. These symbols are non-terminal:
(a) The first capital letters of the alphabet, such as A, B, C.
(b) The letter S which, when it appears, is generally the initial symbol.
(c) Italicized and lowercase names, such as expr or instr.
(d) When talking about programming constructs, capital letters can be used to represent non-terminals. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

3. The last capital letters of the alphabet, such as X, Y, Z, represent grammatical symbols; that is, they can be non-terminal or terminal.

4. The last lowercase letters of the alphabet, such as u, v,..., z, represent strings of terminals (possibly empty).

5. The lowercase Greek letters α, β, γ, for example, represent strings (possibly empty) of grammar symbols. Hence, a generic output can be written as A → α, where A is the header and α the body.

6. A set of productions A → α1, A → α2,..., A → αk with a common header A (we will call them productions A), can be written as A → α1 | α2 | ... | αk. We call α1, α2,..., αk the alternatives for A.

7. Unless otherwise indicated, the heading of the first production is the initial symbol.

$$
\begin{array}{lll}
E & \rightarrow & E \quad | \; E - \quad | \\
 & \rightarrow & \quad * \quad | \quad \blacksquare \quad | \\
 & \rightarrow & (\; E \;) \; | \; \mathbf{id}
\end{array}
$$

Notation conventions tell us that E, T, and F are nonterminal, and E is the initial symbol. The rest of the symbols are terminals.

Derivations

Consider the following grammar, with a single nonterminal E, which adds a production E → - E

E → E + E | E ∗ E | - E | (E) | id (4.7)

The production E → - E means that if E denotes an expression, then - E must also denote an expression. The substitution of a single E for - E will be described by writing the following:
E ⇒ −E
which is read as "E derives to - E". The production E → (E) can be applied to substitute any instance of E in any string of grammatical symbols for (E).

We can take a single E and apply productions repeatedly and in any order to obtain a sequence of substitutions. For example:
E ⇒ −E ⇒ - (E) ⇒ - (id)
We call this sequence of substitutions a derivation of - (id) from E.

The string - (id + id) is a sentence of the grammar (4.7), since there is a derivation
E ⇒ −E ⇒ - (E) ⇒ - (E + E) ⇒ - (id + E) ⇒ - (id + id) (4.8)
The strings E, −E, - (E),..., - (id + id) are all phrase forms of this grammar. We write E ⇒ - ∗ (id + id) to indicate that - (id + id) can be derived from E.

Parse trees and derivations
A parse tree is a graphical representation of a derivation that filters the order in which it is they apply productions to replace non-terminals. Each interior node in a parse tree represents a production application. The inner node is labeled with the nonterminal A in the output header; the children of the node are labeled, left to right, by the symbols in the body of the production this A was substituted for during derivation. The leaves of a parse tree are labeled by nonterminals or terminals and, read from left to right, constitute a phrase form, which is called the product or boundary of the tree.

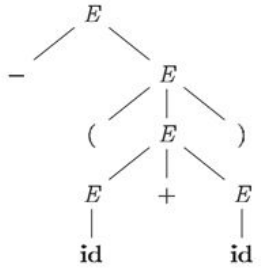


Figura 4.3: Árbol de análisis sintáctico para $-(\mathbf{id} + \mathbf{id})$

Ambiguity
An ambiguous grammar is one that produces more than one derivation from the left, or more than one derivation from the right, for the same sentence.

Example 4.11:
The arithmetic expression grammar (4.3) allows two different left derivations for the statement id + id ∗ id:

$$
\begin{array}{lll}
E & \Rightarrow & E \quad E \\
 & \Rightarrow & \mathbf{id} \quad E \\
 & \Rightarrow & \mathbf{id} \quad E * E \\
 & \Rightarrow & \mathbf{id} \quad \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} \quad \mathbf{id} * \mathbf{id}
\end{array}
\qquad
\begin{array}{lll}
E & \Rightarrow & E * E \\
 & \Rightarrow & E \quad E * E \\
 & \Rightarrow & \mathbf{id} \quad E * E \\
 & \Rightarrow & \mathbf{id} \quad \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} \quad \mathbf{id} * \mathbf{id}
\end{array}
$$

For most parsers, it is convenient that the grammar is unambiguous, otherwise, we cannot uniquely determine which parse tree to select for a statement. In other cases, it is convenient to use carefully chosen ambiguous grammars, along with disambiguation rules, which "discard" unwanted syntactic trees, leaving only one tree for each statement.

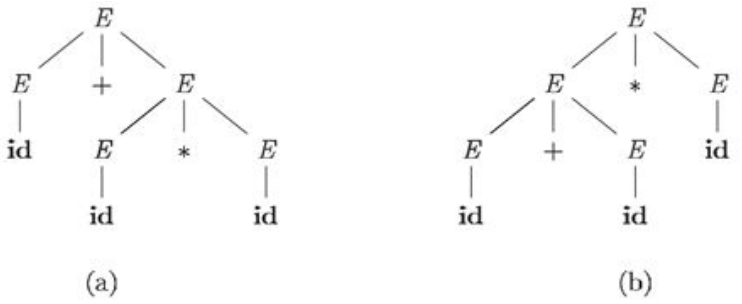


Figura 4.5: Dos árboles de análisis sintáctico para **id+id∗id**

Comparison between context-free grammars and regular expressions
Any structure that can be described by regular expressions can be described by syntax, but not in the opposite way. Or, all conventional language is a language without context, but not the other way around.

For example:
The regular expression (a | b) ∗ abb and the following grammar:

$$
\begin{array}{lll}
A_0 & \rightarrow & aA_0 \mid bA_0 \mid aA_1 \\
A_1 & \rightarrow & bA_2 \\
A_2 & \rightarrow & bA_3 \\
A_3 & \rightarrow & \epsilon
\end{array}
$$

We can mechanically construct a grammar to recognize the same language as a non-deterministic finite automaton (NFA), by means of the following construction:

1. For each state i of the AFN, create a non-terminal Ai.
2. If state i has a transition to state j with input a, add the output Ai → aAj. If state i goes to state j with the input, add the production Ai → Aj.

3. If i is an accepting state, add Ai →.

4. If i is the initial state, make Ai the initial symbol of the grammar.

Writing a grammar

Grammars are capable of describing almost most of the syntax of programming languages. The requirement that identifiers must be declared before use cannot be described by a context-free grammar. Therefore, the sequences of tokens that a parser accepts form a superset of the programming language; subsequent stages of the compiler should parse the output of the parser, to ensure that it complies with the rules that the parser does not check.

Comparison between parsing and parsing

1. By separating the syntactic structure of a language into lexical and non-lexical parts, a convenient way is provided to module a compiler's user interface into two components of a manageable size.
2. The lexical rules of a language are often quite simple, and to describe them we do not need a notation as powerful as grammars.
3. In general, regular expressions provide a more concise and easy-to-understand notation for tokens compared to grammars.
4. More efficient lexical analyzers can be built automatically from regular expressions, compared to arbitrary grammars.

There are no firm guidelines on what to put in the lexical rules, as opposed to the syntactic rules.

Morales Carrillo Gerardo

EXAM

Chapter 2, exercise 9

Algorithm to Remove Left Recursion with an example:

Suppose we have a grammar which contains left recursion:

S -> S a / S b / c / d

Check if the given grammar contains left recursion, if present then separate the production and start working on it.

In our example,

S -> S a / S b / c / d

Enter a new nonterminal and write it at the last of every terminal. We produce a new nonterminal S'and write new production as,

S -> cS '/ dS'

Write newly produced nonterminal in LHS and in RHS it can either produce or it can produce new production in which the terminals or non terminals which followed the previous LHS will be replaced by new nonterminal at last.

S '->? / aS '/ bS'

So after conversion the new equivalent production is

S -> cS '/ dS'

S '->? / aS '/ bS'

Indirect Left Recursion:

A grammar is said to have indirect left recursion if, starting from any symbol of the grammar, it is possible to derive a string whose head is that symbol.

For example,

A -> Br

B -> Cd

C -> At

Where A, B, C are non-terminals and r, d, t are terminals.

Here, starting with A, we can derive A again on substituting C to B and B to A.

Algorithm to remove Indirect Recursion with help of an example:

A1 -> A2 A3

A2 -> A3 A1 / b

A3 -> A1 A1 / a

Where A1, A2, A3 are non terminals and a, b are terminals.

Identify the productions which can cause indirect left recursion. In our case,

A3 - & gt A1 A1 / a

Substitute its production at the place the terminal is present in any other production substitute A1– & gt A2 A3 in production of A3. A3 -> A2 A3 A1.

Now in this production substitute A2– & gt A3 A1 / b and then replace this by,

A3 -> A3 A1 A3 A1 / b A3 A1

Now the new production is converted in form of direct left recursion, solve this by direct left recursion method .

Eliminating direct left recursion in the above,

A3 -> a | b A3 A1 | aA '| b A3 A1A '

A' -> A1 A3 A1 | A1 A3 A1A '

The resulting grammar is then:

A1 -> A2 A3

A2 -> A3 A1 | b

A3 -> a | b A3 A1 | aA '| b A3 A1A '

A' -> A1 A3 A1 | A1 A3 A1

Chapter 3, exercise 1

Let's start with the grammar (step 1 is already done): ´

• E → E + T | T ∗ F | (E) | a | b | Ia | Ib | I0 | I1

• T → T ∗ E | (E) | a | b | Ia | Ib | I0 | I1

• F → (E) | a | b | Ia | Ib | I0 | I1

• I → a | b | Ia | Ib | I0 | I1

For step 2, we introduce new variables and we are left with

the following rules:

A → a, B → b, Z → 0, O → 1

P → +, M → ∗, L → (, R →)

and when replacing we obtain the grammar: ´

E → EPT | TMF | LER | a | b | IA | IB | IZ | IO

T → TPE | LEL | a | b | IA | IB | IZ | IO

F → LER | a | b | IA | IB | IZ | IO

I → a | b | IA | IB | IZ | IO

A → a, B → b, Z → 0, O → 1

P → +, M → ∗, L → (, R →

For the step 3, we replace:

• E → EPT by E → EC1, C1 → PT

• E → TMF, T → TMF by

E → TC2, T → TC2, C2 → MF

• E → LER, T → LER, F → LER by

E → LC3, T → LC3, F → LC3, C3 → ER

The final CNF grammar is: ´

• E → EC1 | TC2 | LC3 | a | b | IA | IB | IZ | IO

• T → TC2 | LC3 | a | b | IA | IB | IZ | IO

• F → LC3 | a | b | IA | IB | IZ | IO

• I → a | b | IA | IB | IZ | IO

• C1 → PT, C2 → MF, C3 → ER

• A → a, B → b, Z → 0, O → 1

• P → +, M → ∗, L → (, R →)