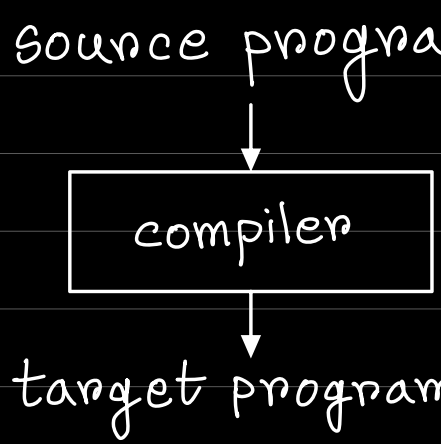
Compiler: a program that can read a program in one language (the source language) and translates into an equivalent program in another language (the target language).

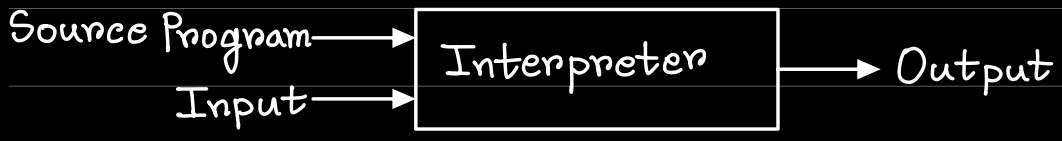
Important role: report any errors in the source program that it detects during the translation process.



If the target program is an executable machine-language program, it then can be called by the user to process input

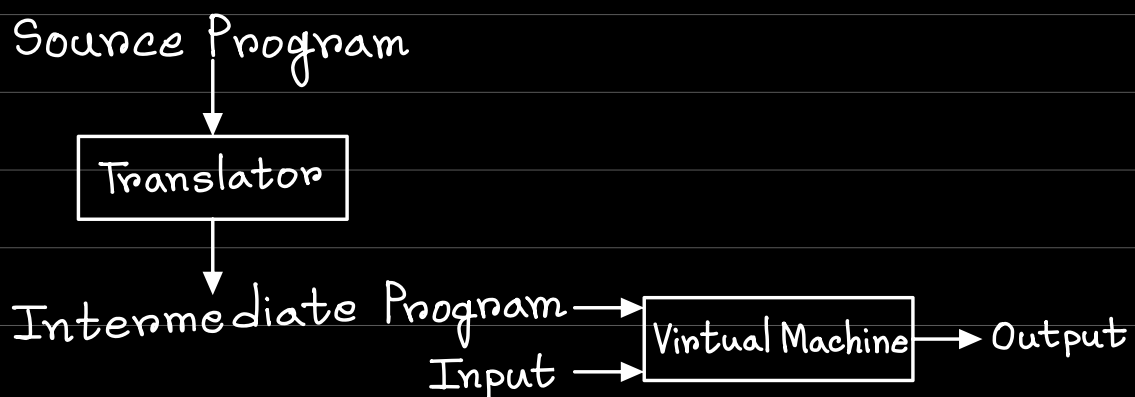
and produce output.

Interpreter: another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

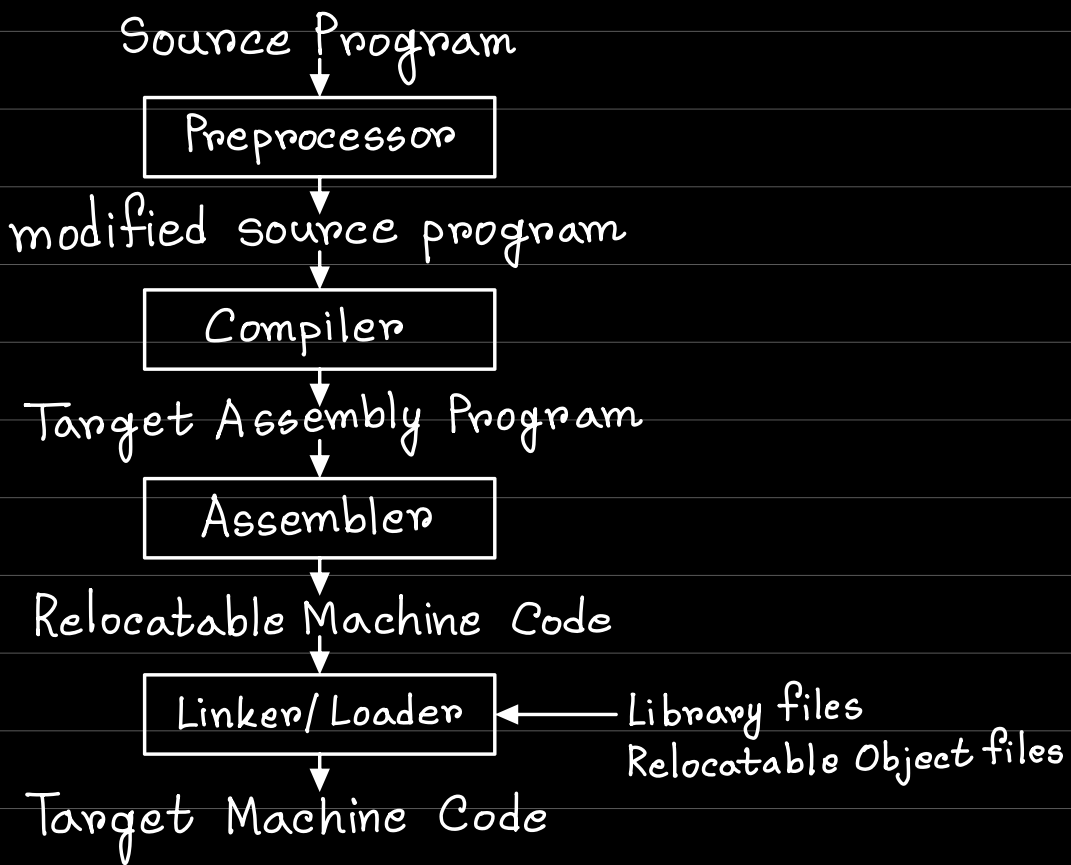


Speed of mapping inputs to outputs:
Compiler > Interpreter
Better at error diagnostic:
Compiler < Interpreter

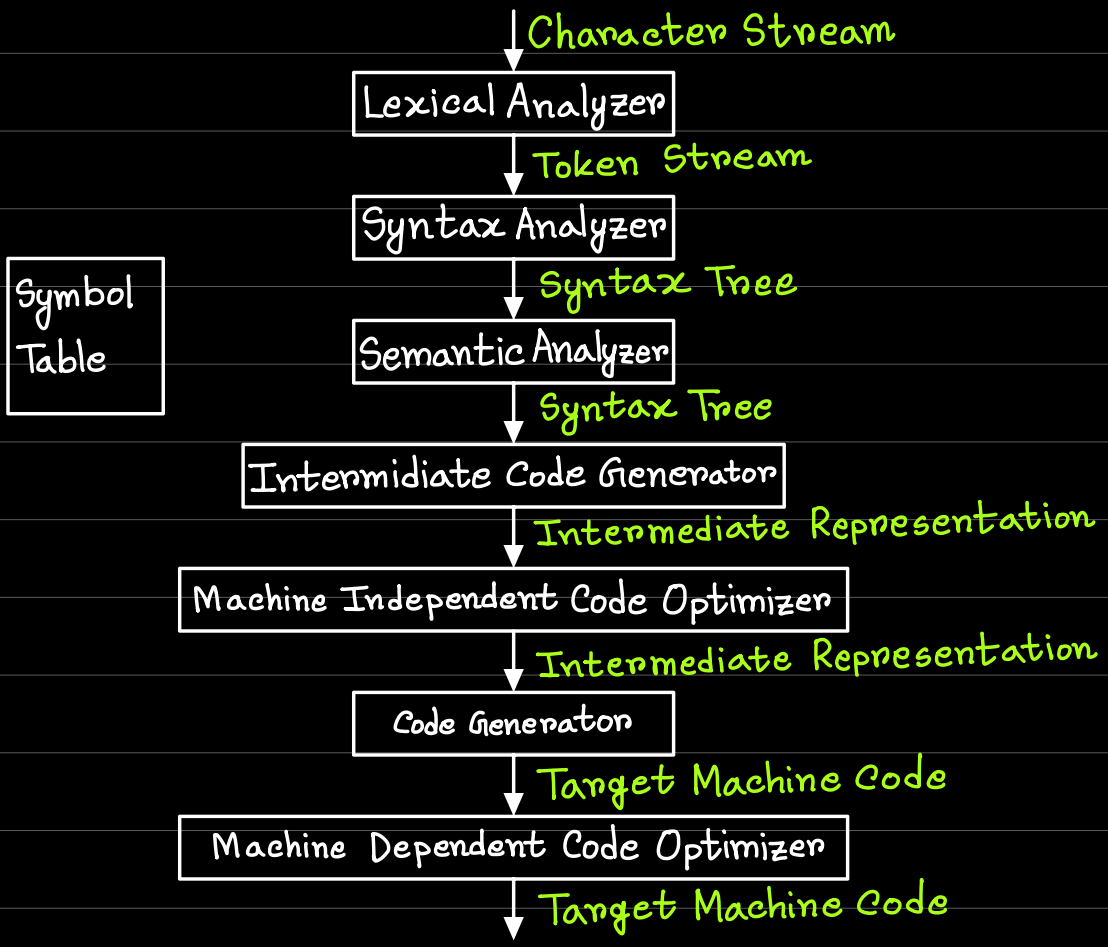
Java Language Processor:



A Language-Processing System:



The Structure of a Compiler



Lexical Analysis: The first phase of a compiler (also known as scanning). The lexical analyzer reads the stream of characters making up the source

program and groups the characters into meaningful sequences called lexemes. For each lexem, the lexical analyzer produces as output a token of the form

<token_name, attribute_value>

that it passes on to the subsequent phase, syntax analysis. Simply, lexical analyzer turns characters into tokens.

position = initial + rate * 60

1. position ⟶ <id, 1>

id: identifier

1: points to the symbol table entry for 'position'.

The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. = ⟶ <=> {assignment symbol}
Needs no attribute-value.

3. initial ⟶ <id, 2>

4. + ⟶ <+> {addition symbol}

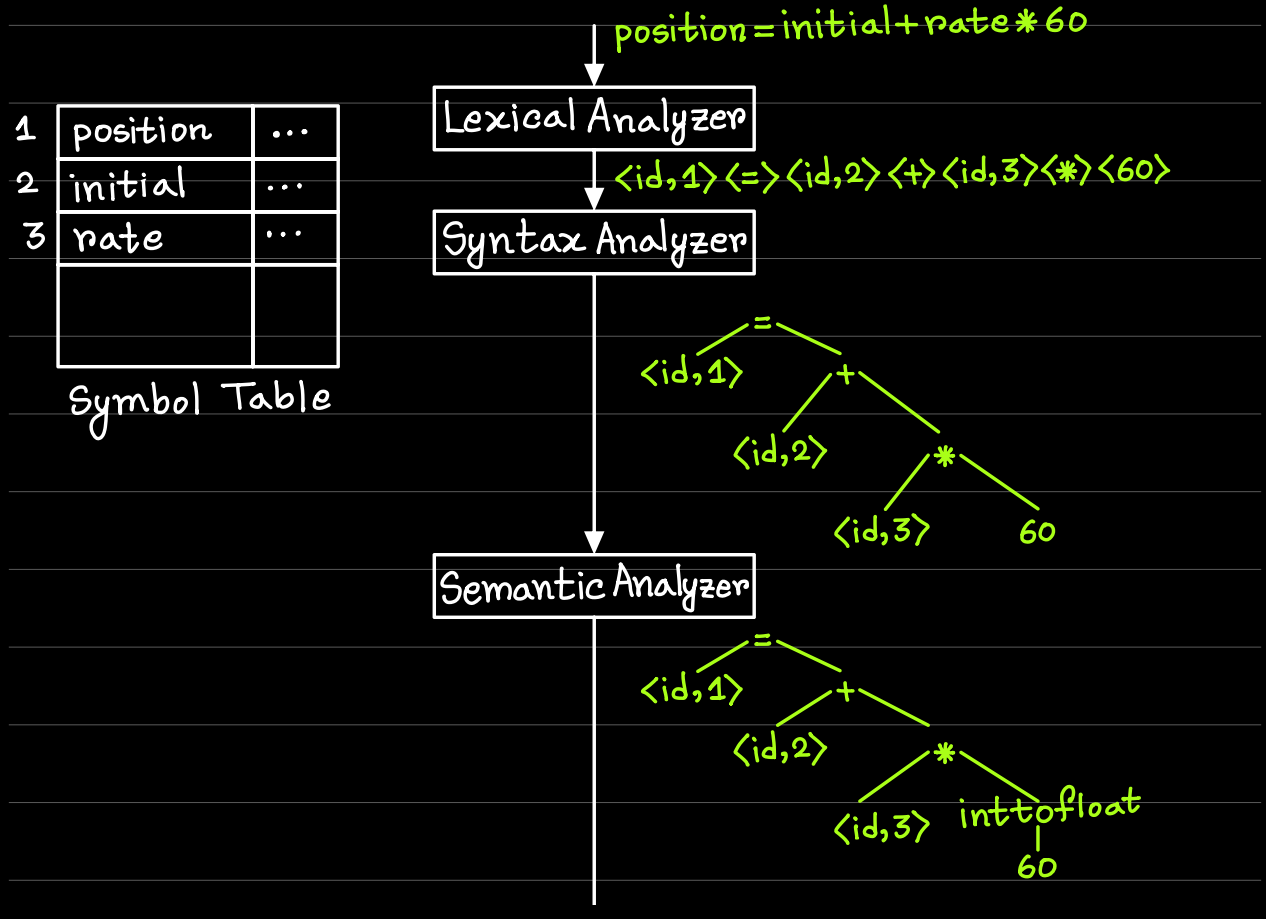
5. rate ⟶ <id, 3>

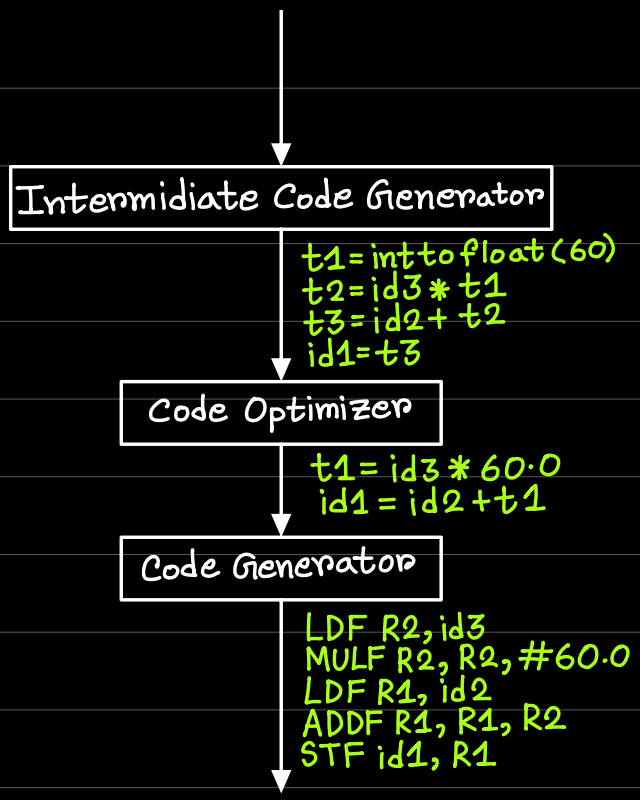
6. * ⟶ <*> {multiplication symbol}

7. 60 ⟶ <60> {We will see this on chapter no. 2 and 3.}

Final result:

<id,1> <=> <id,2> <+> <id,3> <*> <60>





Syntax Analysis: The second phase of the compiler, also known as parsing, checks whether tokens are properly and syntactically organized according to grammar rules. It identifies any errors within the structure. This phase involves creating a parse tree and determining whether the grammar is ambiguous.

Semantic Analysis: Semantic analysis checks logical errors and ensures variables are properly declared. It examines the scope of variables (static or dynamic) and verifies syntax, lexical, and logical components. This phase uses Syntax-Directed Translation (SDT), adding actions based on the grammar rules obtained from the parse tree.

Intermediate Code Generation: Intermediate code is a crucial step in the compilation process, acting as a bridge between the source code and the target machine code. It provides a machine-independent representation, simplifying code optimization and generation for various architectures.

Three-Address Code: A popular form of intermediate code is three-address code, consisting of a sequence of instructions with at most one operator per instruction. Each instruction typically has three operands: a result variable and two source operands. This format facilitates code analysis and optimization.

Code Optimization: Code optimization is a crucial phase in the compilation process that aims to improve the quality of the generated target code. It involves analyzing and transforming the intermediate code to enhance its efficiency, reduce its size, or lower its power consumption.

Symbol Table Management: A symbol table is a crucial data structure used by compilers to store information about identifiers (variables, functions, etc.) used in a program. It helps the compiler keep track of various attributes associated with each identifier, such as:

1. Name: The identifier's name.
2. Type: The data type of the identifier (e.g., integer, float, string).
3. Scope: The region of the program where the identifier is valid.
4. Memory Location: The address where the identifier's value is stored.
5. Attributes: Additional information specific to the identifier, like function parameters, return type, etc.

Key Operations on Symbol Tables:

1. Insertion: Adding a new identifier and its attributes to the table.
2. Lookup: Searching for an existing identifier to retrieve its information.
3. Deletion: Removing an identifier from the table when it goes out of scope.

Symbol Table Data Structures: Various data structures can be used to implement symbol tables, each with its own advantages and disadvantages:

1. Hash Tables: Efficient for both insertion and lookup, but can have collisions.
2. Binary Search Trees: Efficient for both insertion and lookup, but can become unbalanced in certain cases.
3. Tries: Efficient for prefix-based searches, but can be memory-intensive.

The choice of data structure depends on the specific requirements of the compiler, such as the frequency of insertions, deletions, and lookups, as well as the size of the symbol table. By effectively managing the symbol table, compilers can ensure correct type checking, memory allocation, and code generation.

Compiler Phases: Analysis and Synthesis A compiler is a complex program that translates source code written in a high-level language into machine code that can be executed by a computer. It typically involves two main phases:

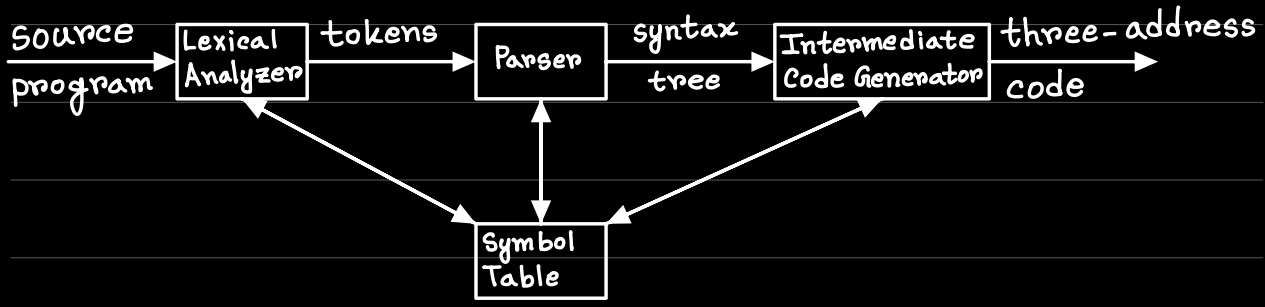
1. Analysis Phase:
- Lexical Analysis: Breaks the source code into meaningful tokens (keywords, identifiers, operators, etc.).
- Syntax Analysis (Parsing): Checks the grammatical structure of the token stream using a formal grammar, often represented as a context-free grammar (CFG).
- Semantic Analysis: Verifies the semantic correctness of the program, including type checking, variable scoping, and flow control analysis.
- Intermediate Code Generation: Creates an intermediate representation of the program, which is a machine-independent form that is easier to optimize and translate into target code.

2. Synthesis Phase:
- Code Optimization: Analyzes the intermediate code to identify optimization opportunities, such as removing redundant code, improving instruction scheduling, and reducing memory usage.
- Code Generation: Translates the optimized intermediate code into the target machine code, which can be executed by the computer.

Context-Free Grammars (CFGs): CFGs are used to formally define the syntax of a programming language. They consist of a set of production rules that specify how language constructs can be derived from simpler ones. CFGs are essential for parsing, as they provide a framework for recognizing and analyzing the structure of source code.

Syntax-Directed Translation: Syntax-directed translation is a technique that uses the grammar of a language to guide the translation process. It involves associating semantic actions with the production rules of the grammar. These actions are executed during parsing to perform tasks like building the symbol table, generating intermediate code, and checking for semantic errors.

Postfix Notation: Postfix notation is a way of writing expressions where the operators appear after their operands. For example, the infix expression a + b * c can be written in postfix notation as a b c * +. Postfix notation is often used in intermediate code generation as it simplifies the process of generating machine code.



The Role of the lexical analyzer: The lexical analyzer, often referred to as the scanner, is the initial phase of a compiler. Its primary role is to break down the source code into meaningful units called lexemes. These lexemes are then categorized and classified into tokens, which are passed on to the subsequent phase, the parser.

Key Tasks of a Lexical Analyzer:

- Read Input Characters: The analyzer reads the source code character by character.
- Identify Lexemes: It groups characters into meaningful units, such as identifiers, keywords, operators, and literals.
- Classify Tokens: Each lexeme is assigned a token type, which indicates its role in the language (e.g., KEYWORD, IDENTIFIER, OPERATOR, LITERAL).
- Symbol Table Interaction: The analyzer may interact with the symbol table to:
- Insert Identifiers: Add new identifiers to the symbol table.
- Retrieve Information: Query the symbol table for information about previously encountered identifiers.

The Lexical Analyzer-Parser Interaction: Typically, the parser controls the lexical analysis process. It sends a request to the lexical analyzer to provide the next token. The lexical analyzer then reads characters from the input, identifies the next lexeme, classifies it as a token, and returns it to the parser.

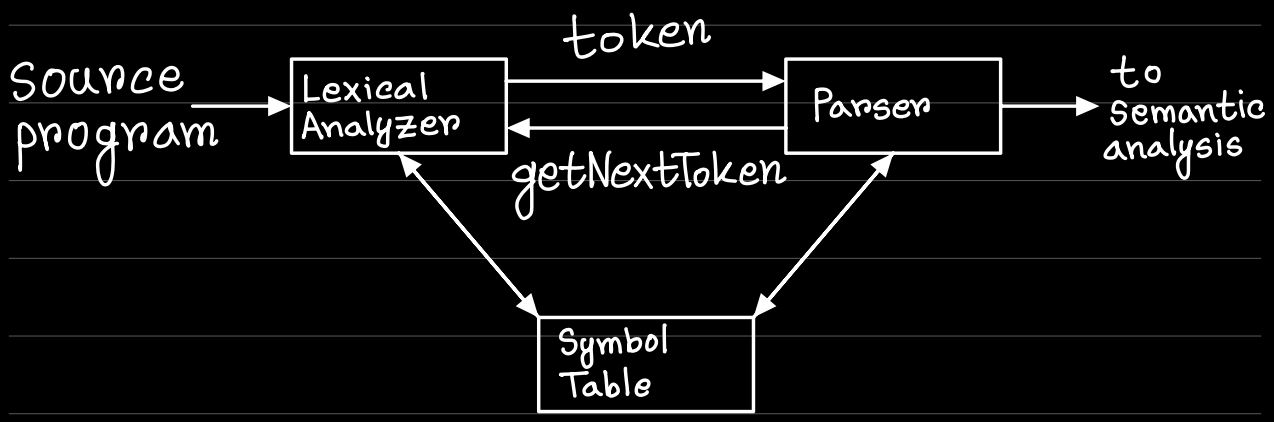
Consider the following C code snippet:

```
int main() {
    int x = 10;
    return x;
}
```

The lexical analyzer would break down this code into the following tokens:

```
KEYWORD int
KEYWORD main
PUNCTUATION (
PUNCTUATION )
PUNCTUATION {
KEYWORD int
IDENTIFIER x
OPERATOR =
LITERAL 10
PUNCTUATION ;
KEYWORD return
IDENTIFIER x
PUNCTUATION ;
PUNCTUATION }
```

These tokens, along with their associated token types, are then passed to the parser for further analysis.



1) Tokenization
2) Give Error Messages
3) Eliminate Comments, White Space (Tab, Blank Space, New Line)

- 1. Exceeding Length
  1. Variable name, funcion name, and values are too large.
- 2. Unmatched String
  2. Forget to close string.
- 3. Illegal Characters
  3. Wrong characters in code.

Tokens

- Identifier (x, y, a, b, ...)
- Separator ({, }, (,), ...)
- Keyword (int, main, if, else, return, print, ...)
- Operator (<, >, +, -, *, /, %, =, ...)
- Constants (literals) (20, 30, True, False, ...)
- Special Characters

C Code:

```
int main() {
    /* find max of a and b */
    int a = 20, b = 30;
    if (a<b)
        return (b);
    else
        return (a);
}
```

Total 32 tokens.

```
printf("i=%d, &i=%x", i, &i);
```

Single String counted as 1.

Total 10 tokens.

Here, we use finite automata (DFA or NFA). Error in code does not matter in lexical analysis.

Practice 1:

```
int main)(
{
    x = y+z;
    int x, y, z;
    Printf("Sum %d %d", x);
{
```

Total 26 tokens.

Practice 2:

Here, ++, +=, --, -=, and == act like single tokens.

```
main(){
a=b+++----++==;
Printf("%d%d, a, b);
}
```

From this quotation, we will stop counting including it because there is no next quotation.

Total 17 tokens.

Practice 3:

```
main(){
int a=10;
char b="abc";
in t c =30;
ch ar d ="xyz";
in /* Comment */t m=40.5;
}
```

Even there is a gap between {in and t}/ {ch and ar}, we will count them separately.

We will remove comment even if there is an error.

Total 33 tokens.

Additional Roles of the Lexical Analyzer: Beyond its primary role of tokenizing the source code, the lexical analyzer can perform several additional tasks to enhance the overall compilation process:

1. Preprocessing:
- Comment Removal: Eliminates comments from the source code.
- Whitespace Handling: Compresses multiple whitespace characters into a single space or removes them entirely.
- Macro Expansion: If the language supports macros, the lexical analyzer can expand them into their corresponding code.

2. Error Handling:
- Line Number Tracking: Keeps track of the current line number to provide accurate error messages.
- Error Reporting: Generates informative error messages that highlight the location of the error in the source code.
- Error Recovery: Implements strategies to recover from errors and continue the compilation process, such as skipping tokens or inserting missing tokens.

3. Source Code Preservation:
- Source Code Copying: Creates a copy of the source code with error messages inserted at the appropriate locations.
- Formatting: Preserves the original formatting of the source code to improve readability of error messages.

Two-Phase Lexical Analysis: In some compilers, the lexical analysis process is divided into two phases:

1. Scanning:
- Simple Tasks: Handles tasks like removing comments, whitespace, and string literals.
- Input Buffering: Manages the input buffer to optimize character access.

2. Lexical Analysis:
- Tokenization: Identifies and classifies tokens, such as keywords, identifiers, operators, and literals.
- Symbol Table Interaction: Interacts with the symbol table to store information about identifiers.
- Error Handling: Detects and reports lexical errors.

The Benefits of Separating Lexical Analysis and Parsing: The division of the analysis phase into lexical analysis and parsing offers several advantages:

1. Simplicity of Design:
- Focused Tasks: By separating concerns, each phase can focus on its specific task, leading to simpler implementations.
- Reduced Parser Complexity: The parser can assume a clean input stream without worrying about comments, whitespace, and other low-level details.
- Cleaner Language Design: A clear distinction between lexical and syntactic elements can lead to a more elegant and maintainable language design.

2. Improved Compiler Efficiency:
- Specialized Techniques: The lexical analyzer can employ specialized techniques for efficient character input and tokenization, such as buffering and lookahead.
- Optimized Tokenization: The lexical analyzer can optimize the process of recognizing and classifying tokens, leading to faster overall compilation times.

3. Enhanced Compiler Portability:
- Platform-Specific Details: The lexical analyzer can handle platform-specific input/output operations and character encoding issues, making the parser more portable.
- Language-Specific Lexical Rules: The lexical analyzer can be tailored to the specific lexical rules of the language, while the parser remains relatively language-independent.

In essence, the separation of lexical analysis and parsing leads to a more modular, efficient, and portable compiler design.

Let’s break down the terms ‘tokens’, ‘patterns’, and ‘lexemes’ in the context of lexical analysis.

Tokens: Tokens are like little pieces of information that make up the language. They’re made up of two parts: a name and an optional value. The name is like a label that tells us what kind of token it is, like a keyword or an identifier. The value is just the actual characters that make up the token. We’ll often write the name of a token in boldface.

Patterns: Patterns are like rules that tell us how the tokens in a language can be formed. For example, the pattern for a keyword is just the sequence of characters that make up the keyword. For identifiers and some other tokens, the pattern is a bit more complex and can match many different strings.

Lexemes: Lexemes are like the actual tokens that are recognized by the lexical analyzer. They’re the sequences of characters in the source program that match the pattern for a token and are identified by the lexical analyzer as an instance of that token.

Tokenization in Compiler Design

A compiler breaks down source code into smaller units called tokens. These tokens represent the building blocks of the language, such as keywords, identifiers, operators, and literals.

Key Token Classes:

Keywords: Reserved words with specific meanings (e.g., if, else, for).
Operators: Symbols that perform operations (e.g., +, -, *, /, =, <, >, !).
Identifiers: Names given to variables, functions, or other entities (e.g., x, y, myFunction).
Literals: Constant values (e.g., numbers like 42 or strings like "Hello, world!").
Punctuators: Special symbols used to structure code (e.g., (,), {, }, ;, ,).

Example:

In the C statement printf("Total = %d\n", score);:

printf and score are identifiers.
"Total = %d\n" is a literal string.
(and) are punctuators.
; is a punctuator.

In essence, tokenization is the process of breaking down source code into meaningful units that the compiler can further process.

Token Attributes

When a lexical analyzer encounters a lexeme that matches multiple token patterns, it needs to provide additional information to the subsequent phases of the compiler. This information is often referred to as a token attribute.

Common Token Attributes:

Identifier:
Pointer to the symbol table entry
Lexeme
Type
Location

Number:
Integer or floating-point value
String representation

String Literal:
String value

Keyword:
No specific attribute, often implied by the token name

Example:

For the Fortran statement E = M * C**2, the token stream with attributes might look like this:

<id, pointer to E>
<assign-op>
<id, pointer to M>
<mult-op>
<id, pointer to C>
<exp-op>
<number, integer value 2>

Why Token Attributes are Important:

Semantic Analysis: Helps in type checking, variable scoping, and other semantic checks.
Code Generation: Provides information for generating machine code, such as variable addresses, constant values, and function calls.
Error Reporting: Helps in providing accurate and informative error messages, pointing to the specific location and cause of the error.
By associating attributes with tokens, the compiler can effectively process and analyze the source code, ultimately producing efficient and correct machine code.

Lexical Errors and Recovery

Lexical Errors:

Occur when the lexical analyzer encounters a sequence of characters that doesn't match any valid token pattern.
Can be caused by typos, missing characters, or incorrect syntax.
The lexical analyzer cannot definitively determine the error without context from the parser or semantic analyzer.

Error Recovery Strategies:

Panic Mode Recovery:
Deletes characters from the input until a valid token is found.
Simple but can lead to cascading errors.

Single-Character Transformations:
Tries to correct the error by deleting, inserting, replacing, or transposing a single character.
More sophisticated than panic mode but still limited.

More Complex Transformations:
Considers multiple transformations to correct the input.
Can be computationally expensive and may not always yield the correct result.

Example:

Consider the following C code with a typo:

```
int x = 10;
x = x + y; // Typo: 'y' should be '10'
```

The lexical analyzer might encounter the token y when it's not declared. In this case, it could:

Panic Mode: Delete y and continue, leading to a syntax error later.
Single-Character Transformation: Try inserting a digit before y to form a number.
More Complex Transformation: Analyze the context and suggest possible corrections, such as replacing y with 10.
In practice, a combination of these strategies is often used, with the choice depending on the severity of the error, the complexity of the language, and the desired level of error recovery.

Exercise 3.1.1: C++ Lexemes

```
float limitedSquare(x) float x {
   /* returns x-squared, but never more than 100 */
   return (x <= -10.0 || x >= 10.0) ? 100 : x * x;
}
```

Lexemes and their associated values:

Keywords: float, return
Identifiers: limitedSquare, x
Operators: (,), {, }, =, <, <=, >, >=, ||, ?, :, *, ;
Literals: 100, 10.0, -10.0

Exercise 3.1.2: HTML Lexemes

```
HTML
Here is a photo of <B>my house</B>:
<P><IMG SRC="house.gif"><BR>
See <A HREF="morePix.html">More Pictures</A> if you
liked that one. <P>
```

Lexemes and their associated values:

Keywords: <B>, </B>, <P>, <IMG>, SRC=, <BR>, <A>, HREF=, </A>
Identifiers: my house, house.gif, More Pictures, morePix.html
Punctuators: <, >, ", , \n
Note:

In HTML, the tags themselves (like <B>) are considered keywords, as they have specific meanings.
The text within tags (like my house) can be considered identifiers or literals, depending on how they are processed by the HTML parser.
Attributes like SRC and HREF are often treated as keywords, while their values (like house.gif) are considered literals.
The specific categorization of lexemes can vary depending on the parser and the desired level of granularity in the lexical analysis.

Specification of tokens:
Regular expressions are a powerful tool for defining patterns of text. They are used to specify the exact format of tokens in a programming language. By using regular expressions, we can create a precise and efficient lexical analyzer that can accurately identify and categorize tokens in source code.

The process involves:
Defining Regular Expressions: Writing regular expressions to describe the patterns of tokens.
Generating Automata: Converting these regular expressions into finite automata, which are mathematical models for recognizing patterns.
Building the Lexical Analyzer: Using the generated automata to create a lexical analyzer that can scan the source code and identify tokens based on the defined patterns.

Buffer Space and Lookahead
While most modern languages have relatively short lexemes, certain edge cases can arise:

Long Character Strings:
Languages like Java often allow multi-line strings, which can potentially be very long.
To handle this, lexical analyzers can break down long strings into smaller chunks, each within the buffer limit.
Concatenation operators or special escape sequences can be used to combine these chunks.

Arbitrary Lookahead:
Some languages, like PL/I, allow identifiers to be the same as keywords.
This can lead to ambiguity, as the lexical analyzer may need to look ahead to determine the correct token.
To handle this, modern languages often reserve keywords.
If not, the lexical analyzer can treat ambiguous identifiers as such and let the parser resolve the ambiguity based on context and symbol table information.

Buffer Size Considerations:
A balance must be struck between buffer size and memory usage.
A larger buffer can handle longer lexemes but consumes more memory.
A smaller buffer may be sufficient for most cases but can lead to performance issues if many long lexemes are encountered.
By carefully considering these factors and employing appropriate techniques, lexical analyzers can effectively handle various input scenarios and avoid buffer overflow issues.

Alphabet and Symbols
- Alphabet: A finite set of symbols. Examples include letters, digits, and punctuation.
  - Binary Alphabet: The set {0, 1}.
  - ASCII: An important alphabet used in many software systems.
  - Unicode: An extensive alphabet including around 100,000 characters from various world alphabets.

Strings
- String: A finite sequence of symbols from an alphabet.
  - Often referred to as a "sentence" or "word" in language theory.
  - Length of a String (|s|): The number of symbols in the string.
    - Example: "banana" is a string of length 6.
    - Empty String (ε): A string of length zero.

Languages
- Language: Any countable set of strings over a fixed alphabet.
  - Includes abstract languages like ∅ (empty set) and {ε} (set containing only the empty string).
  - Examples:
    - All syntactically well-formed C programs.
    - All grammatically correct English sentences (though difficult to specify exactly).
  - Note: The definition of "language" doesn't require assigning meaning to the strings. Defining the "meaning" of strings is a topic covered in Chapter 5.

Sentinels at the End of Each Buffer

Code Explanation
Here's the code for handling end-of-file (EOF) characters and reloading buffers in a lexical analyzer:

```
switch (*forward++) {
  case eof:
    if (forward is at end of the first buffer) {
      reload second buffer;
      forward = beginning of the second buffer;
    } else if (forward is at end of the second buffer) {
      reload first buffer;
      forward = beginning of the first buffer;
    } else {
      // eof within a buffer marks the end of input
      terminate lexical analysis;
    }
    break;
  // Cases for other characters
}
```

Key Points
- EOF Handling: The `switch` statement checks if the `forward` pointer encounters an EOF character.
- Buffer Reloading: If `forward` is at the end of the first buffer, it reloads the second buffer and moves `forward` to the beginning of the second buffer. Similarly, if `forward` is at the end of the second buffer, it reloads the first buffer and resets `forward`.
- Termination: If an EOF character is found within a buffer, it marks the end of input, and lexical analysis is terminated.
- Other Characters: The code includes cases for handling other characters, though they are not explicitly shown here.

Lookahead Code with Sentinels (Figure 3.5)
This code allows the lexical analyzer to handle lookahead by using sentinels. Sentinels are special characters placed at the end of buffers to signal EOF.

Implementing Multiway Branches
- Multiway Branch Concept: The `switch` statement can handle multiple cases based on the input character.
- Efficiency: The order of case statements does not impact performance significantly. In practice, a multiway branch can be executed in one step by jumping to an address found in an array of addresses indexed by characters.

Summary
1. Sentinels and Buffer Management: Ensuring smooth transition and reloading of buffers during lexical analysis.
2. Multiway Branches: Efficient handling of different characters using a `switch` statement.

Implementing Multiway Branches

Key Points
- Efficiency of `switch` Statement:
  - It may seem like the `switch` statement in Fig. 3.5 requires many steps to execute.
  - The placement of the `case eof` (end-of-file) does not affect the efficiency.

- Order of Cases:
  - The order in which cases are listed doesn't impact performance significantly.

- Multiway Branch Mechanism:
  - In practice, a multiway branch based on the input character is executed efficiently.
  - This is done in one step by jumping to an address found in an array of addresses, indexed by the characters.
  - This method ensures that the correct case is executed quickly, making the process faster and more efficient.

Terms for Parts of Strings

1. Prefix
   - A prefix of a string (s) is any string obtained by removing zero or more symbols from the end of (s).
   - Examples: For the string "banana," the prefixes are "ban," "banana," and ε (the empty string).

2. Suffix
   - A suffix of a string (s) is any string obtained by removing zero or more symbols from the beginning of (s).
   - Examples: For the string "banana," the suffixes are "nana," "banana," and ε.

3. Substring
   - A substring of (s) is obtained by deleting any prefix and any suffix from (s).
   - Examples: For the string "banana," the substrings include "banana," "nan," and ε.

4. Proper Prefixes, Suffixes, and Substrings
   - Proper prefixes, suffixes, and substrings of a string (s) are those prefixes, suffixes, and substrings that are not ε and are not equal to (s) itself.
   - Example: For "banana," proper prefixes are "ban," "bana," "banan," but not "banana" or ε.

5. Subsequence
   - A subsequence of (s) is any string formed by deleting zero or more (not necessarily consecutive) positions of (s).
   - Examples: For "banana," "baan" is a subsequence.

String Concatenation and Exponentiation

1. Concatenation:
   - Definition: If x and y are strings, then the concatenation of x and y, denoted xy, is the string formed by appending y to x.
   - Example: If x = "dog" and y = "house," then xy = "doghouse."
   - Identity Property: The empty string (ε) is the identity under concatenation. This means for any string s:
     - εs = sε = s.

2. Exponentiation of Strings:
   - Think of concatenation as a product.
   - Definition:
     - Define s^0 to be ε (the empty string).
     - For all i > 0, define s^i as s^(i-1)s.
   - Examples:
     - s^1 = s (since s^0 * s = s).
     - s^2 = ss.
     - s^3 = sss.
     - And so on.

Operations on Languages

In lexical analysis, the most important operations on languages are union, concatenation, and closure, which are defined formally in Fig. 3.6. Union is the familiar operation on sets. The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them. The (Kleene) closure of a language L, denoted L*, is the set of strings you get by concatenating L zero or more times. Note that L^0, the "concatenation of L zero times," is defined to be {ε}, and inductively, L^i is L^(i-1)L. Finally, the positive closure, denoted L+, is the same as the Kleene closure, but without the term L^0. That is, ε will not be in L+ unless it is in L itself.

OPERATION:
1. Union of L and M
2. Concatenation of L and M
3. Kleene closure of L
4. Positive closure of L

DEFINITION AND NOTATION:
1. L ∪ M = {s | s is in L or s is in M}
2. LM = {st | s is in L and t is in M}
3. L* = Union of L^i for all i >= 0
4. L+ = Union of L^i for all i > 0

Figure 3.6: Definitions of operations on languages

Example 3.3: Let L be the set of letters {A, B, ..., Z, a, b, ..., z} and let D be the set of digits {0, 1, ..., 9}. We can think of L and D in two essentially equivalent ways. One way is that L and D are the alphabets of uppercase and lowercase letters and digits, respectively. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from L and D using the operations from Figure 3.6:

1. L ∪ D is the set of letters and digits. Strictly speaking, it is the language with 62 strings of length one, each of which is either a letter or a digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. L* is the set of all strings of letters, including ε, the empty string.
5. L(L ∪ D)* is the set of all strings of letters and digits beginning with a letter.
6. D+ is the set of all strings of one or more digits.

Regular Expressions

Suppose we wanted to describe the set of valid C identifiers. It is almost exactly the language described in item (5) above; the only difference is that the underscore is included among the letters. In Example 3.3, we were able to describe identifiers by giving names to sets of letters and digits and using the language operators union, concatenation, and closure.

This process is so useful that a notation called regular expressions has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if letter- is established to stand for any letter or the underscore, and digit- is established to stand for any digit, then we could describe the language of C identifiers by:
letter- (letter- ∪ digit-)*
The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of," and the juxtaposition of letter- with the remainder of the expression signifies concatenation.

The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet ∑ and the languages that those expressions denote.

BASIS:
1. ε is a regular expression, and L(ε) is {ε}, that is, the language whose sole member is the empty string.
2. If a is a symbol in ∑, then a is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with a in its one position.

INDUCTION:
1. (r ∪ s) is a regular expression denoting the language L(r) ∪ L(s).
2. (rs) 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 denoting L(r). This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:
a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) ∪ has lowest precedence and is left associative.

Under these conventions, for example, we may replace the regular expression (a ∪ (b * c)) by a ∪ b*c. Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

Example 3.4:
Let ∑ = {a, b}.
1. The regular expression a ∪ b denotes the language {a, b}.
2. (a ∪ b)(a ∪ b) denotes {aa, ab, ba, bb}, the language of all strings of length two over the alphabet ∑. Another regular expression for the same language is aa ∪ ab ∪ ba ∪ bb.
3. a* denotes the language consisting of all strings of zero or more a's, that is, {ε, a, aa, aaa, ...}.
4. (a ∪ b)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: {ε, a, b, aa, ab, ba, bb, aaa, ...}. Another regular expression for the same language is (a*b*)*.
5. a ∪ a*b denotes the language {a, b, ab, aab, aaab, ...}, that is, the string a and all strings consisting of zero or more a's and ending in b.

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. For instance, (a ∪ b) = (b ∪ a). There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent.

Figure 3.7: Algebraic laws for regular expressions

Law	Description
r ∪ s = s ∪ r	∪ is commutative
r ∪ (s ∪ t) = (r ∪ s) ∪ t	∪ is associative
r(st) = (rs)t	Concatenation is associative
r(s ∪ t) = rs ∪ rt	Concatenation distributes over ∪
εr = rε = r	ε is the identity for concatenation
r* = (r ∪ ε)*	ε is guaranteed in a closure
r** = r*	* is idempotent

Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If ∑ 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, not in ∑ and not the same as any other of the d's.
2. Each ri is a regular expression over the alphabet ∑ ∪ {d1, d2, ..., di-1}.

By restricting ri to ∑ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over ∑ alone, for each ri. We do so by first replacing uses of d1 in r2 (which cannot use any of the d's except for d1), then replacing uses of d1 and d2 in r3 by r1 and (the substituted) r2, and so on. Finally, in rn we replace each di, for i = 1, 2, ..., n-1, by the substituted version of ri, each of which has only symbols of ∑.

Example 3.5: C identifiers
C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

letter → A ∪ B ∪ ... ∪ Z ∪ a ∪ b ∪ ... ∪ z
digit → 0 ∪ 1 ∪ ... ∪ 9
id → letter (letter ∪ digit)*

Example 3.6: Unsigned numbers
Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition is:

digit → 0 ∪ 1 ∪ ... ∪ 9
digits → digit digit*
optionalFraction → . digits ∪ ε
optionalExponent → E (+ ∪ - ∪ ε) digits ∪ ε
number → digits optionalFraction optionalExponent

This is a precise specification for this set of strings. An optionalFraction is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An optionalExponent, if not missing, is the letter E followed by an optional + or - sign, followed by one or more digits. Note that at least one digit must follow the dot, so number does not match 1., but does match 1.0.

Extensions of Regular Expressions

Since Kleene introduced regular expressions with the basic operators for union, concatenation, and Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns. Here we mention a few notational extensions that were first incorporated into Unix utilities such as Lex that are particularly useful in the specification of lexical analyzers. The references to this chapter contain a discussion of some regular-expression variants in use today.

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+ ∪ ε and r+ = rr* = r*r, relate the Kleene closure and positive closure.

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

3. Character classes: A regular expression a1 ∪ a2 ∪ ... ∪ an, where the ai's are each symbols of the alphabet, can be replaced by the shorthand [a1a2...an]. More importantly, when a1, a2, ..., an form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a1-an, that is, just the first and last separated by a hyphen. Thus, [abc] is shorthand for a ∪ b ∪ c, and [a-z] is shorthand for a ∪ b ∪ ... ∪ z.

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

letter → [A-Za-z_]
digit → [0-9]
id → letter (letter ∪ digit)*

The regular definition of Example 3.6 can also be simplified:

digit → [0-9]
digits → digit+
number → digits (. digits)? (E [+-]? digits)?

Exercise 3.3.1:** Consult the language reference manuals to determine (i) the sets of characters that form the input alphabet (excluding those that may only appear in character strings or comments), (ii) the lexical form of numerical constants, and (iii) the lexical form of identifiers for each of the following languages: (a) C (b) C++ (c) C# (d) Fortran (e) Java (f) Lisp (g) SQL.

Exercise 3.3.2: Describe the languages denoted by the following regular expressions:
a) a(a|b)*a
b) ((ε|a)b*)*
c) (a|b)*a(a|b)(a|b)
d) a*ba*ba*ba*
e) (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

Exercise 3.3.3: In a string of length n, how many of the following are there?
a) Prefixes
b) Suffixes
c) Proper prefixes
d) Substrings
e) Subsequences

Exercise 3.3.4: Most languages are case-sensitive, so keywords can be written only one way, and the regular expressions describing their lexeme are very simple. However, some languages, like SQL, are case-insensitive, so a keyword can be written either in lowercase, uppercase, or any mixture of cases. Thus, the SQL keyword SELECT can also be written select, Select, or sElEcT, for instance. Show how to write a regular expression for a keyword in a case-insensitive language. Illustrate the idea by writing the expression for "select" in SQL.

Exercise 3.3.5: Write regular definitions for the following languages:
a) All strings of lowercase letters that contain the five vowels in order.
b) All strings of lowercase letters in which the letters are in ascending lexicographic order.
c) Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double quotes (").

d) All strings of digits with no repeated digits. Hint: Try this problem first with a few digits, such as {0,1,2}.
e) All strings of digits with at most one repeated digit.
f) All strings of a's and b's with an even number of a's and an odd number of b's.
g) The set of Chess moves, in the informal notation, such as p-k4 or kbp x qn.
h) All strings of a's and b's that do not contain the substring abb.
i) All strings of a's and b's that do not contain the subsequence abb.

Exercise 3.3.6: Write character classes for the following sets of characters:
a) The first ten letters (up to "j") in either upper or lower case.
b) The lowercase consonants.
c) The digits in a hexadecimal number (choose either upper or lower case for the digits above 9).
d) The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

The following exercises, up to and including Exercise 3.3.10, discuss the extended regular-expression notation from Lex (the lexical-analyzer generator that we shall discuss extensively in Section 3.5). The extended notation is listed in Figure 3.8.

Exercise 3.3.7: Note that these regular expressions give all of the following symbols (operator characters) a special meaning:

\ " . ^ $ [ ] * + ? { } | /

Their special meaning must be turned off if they are needed to represent themselves in a character string. We can do so by quoting the character within a string of length one or more; e.g., the regular expression "**" matches the string **. We can also get the literal meaning of an operator character by preceding it with a backslash. Thus, the regular expression ** also matches the string **. Write a regular expression that matches the string "\.

Exercise 3.3.8: In Lex, a complemented character class represents any character except the ones listed in the character class. We denote a complemented class by using ^ as the first character; this symbol (caret) is not itself part of the class being complemented, unless it is listed within the class itself. Thus, [^A-Za-z] matches any character that is not an uppercase or lowercase letter, and [^^] represents any character but the caret (or newline, since newline cannot be in any character class). Show that for every regular expression with complemented character classes, there is an equivalent regular expression without complemented character classes.

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
"s"	string s literally	`"**"`
.	any character but newline	`a.*b`
^	beginning of a line	`^abc`
$	end of a line	`abc$`
[s]	any one of the characters in string s	`[abc]`
[^s]	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a[1,5]`
r_1r_2	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Figure 3.8: Lex regular expressions

Exercise 3.3.9: The regular expression r{m, n} matches from m to n occurrences of the pattern r. For example, a{1, 5} matches a string of one to five a's. Show that for every regular expression containing repetition operators of this form, there is an equivalent regular expression without repetition operators.

Answer: We can replace the repetition operators with explicit concatenation and union of regular expressions. For r{1, 5}, where m = 1 and n = 5: r{1, 5} can be rewritten as: r ∪ rr ∪ rrr ∪ rrrr ∪ rrrrr.

Exercise 3.3.10: The operator ^ matches the left end of a line, and $ matches the right end of a line. The operator ^ is also used to introduce complemented character classes, but the context always makes it clear which meaning is intended. For example, ^[Aaeiou]*$ matches any complete line that does not contain a lowercase vowel.
a) How do you tell which meaning of ^ is intended?
b) Can you always replace a regular expression using the ^ and $ operators by an equivalent expression that does not use either of these operators?

Answer:
a) To determine which meaning of ^ is intended, examine the context:
- If ^ appears at the start of the regular expression, it indicates the beginning of a line.
- If ^ appears inside square brackets [ ], it indicates a complemented character class.

b) Yes, you can replace expressions using ^ and $ with equivalent expressions that do not use these operators by explicitly specifying the required conditions. For example, for ^r$ (matches r at the start and end of a line), you can write a condition checking that the whole string matches r.

Exercise 3.3.11: The UNIX shell command sh uses the operators in Figure 3.9 in filename expressions to describe sets of file names. For example, the filename expression *.o matches all file names ending in .o; sort1.? matches all filenames of the form sort1.c, where c is any character. Show how sh filename expressions can be replaced by equivalent regular expressions using only the basic union, concatenation, and closure operators.

EXPRESSION	MATCHES	EXAMPLE
's'	string *s* literally	`'\'`
\c	character *c* literally	`\'`
*	any string	`*.o`
?	any character	`sort1.?`
[s]	any character in *s*	`sort1.[cso]`

Figure 3.9: Filename expressions used by the shell command `sh`

Exercise 3.3.12: SQL allows a rudimentary form of patterns in which two characters have special meaning:
- Underscore (_) stands for any one character.
- Percent-sign (%) stands for any string of 0 or more characters.

In addition, the programmer may define any character, say e, to be the escape character, so e preceding _, %, or another e gives the character that follows its literal meaning. Show how to express any SQL pattern as a regular expression, given that we know which character is the escape character.

1. Special Characters:
   - _ (SQL: any one character) → . (Regex: any one character)
   - % (SQL: any string of 0 or more characters) → .* (Regex: zero or more characters)

2. Escape Character:
   - e_ → _
   - e% → %
   - ee → e

3. Steps to Convert:
   - Replace _ with ..
   - Replace % with .*.
   - Handle escape characters to treat the next character as literal.

Example:
- SQL Pattern: e_%e% (with e as escape character)
- Intermediate Pattern: _%%
- Final Regex: .*.*

Syntax Analysis

Introduction:
- Parsing methods are crucial in compilers.
- Covers basic concepts, hand-implementation techniques, and algorithms for automated tools.
- Discusses error recovery in programs with syntactic errors.

Programming Language Syntax:
- Programming languages have precise syntactic rules for well-formed programs.
- Example in C: Programs consist of functions, functions consist of declarations and statements, and statements consist of expressions.

Grammars and Notation:
- Syntax of programming languages can be defined using context-free grammars or BNF (Backus-Naur Form).
- Grammars are beneficial for both language designers and compiler writers:
  - Provide clear, understandable syntactic specifications.
  - Allow automatic construction of efficient parsers for determining the syntactic structure of source programs.
  - Help identify syntactic ambiguities and issues during the design phase.

Benefits of Using Grammars:
- Aid in translating source programs into correct object code.
- Help detect errors.
- Enable iterative development of programming languages by adding new constructs.
- Ensure new constructs integrate easily into the existing grammatical structure.

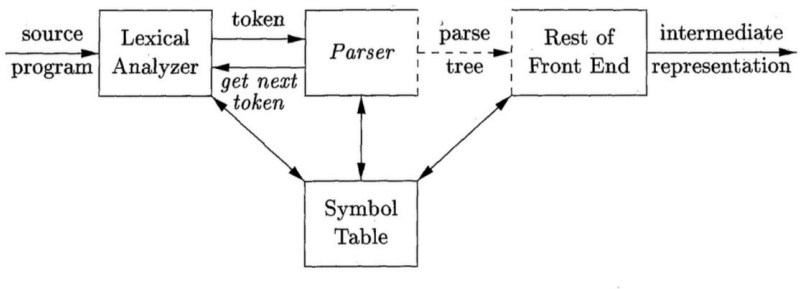


Figure 4.1: Position of parser in compiler model

Parser in Compiler

Introduction:
- Overview of how the parser integrates into a typical compiler.
- Focus on grammars for arithmetic expressions.

Grammars for Expressions:
- Grammars for expressions illustrate core parsing concepts.
- Parsing techniques for expressions can be applied to most programming constructs.

Error Handling:
- Discussion on how the parser handles errors.
- Importance of the parser responding gracefully when input doesn't match its grammar.

Role of the Parser:
- The parser receives a string of tokens from the lexical analyzer.
- Verifies if the token string can be generated by the grammar for the source language.
- Reports syntax errors clearly and recovers from common errors to continue processing the program.

Parse Tree:
- Constructs a parse tree for well-formed programs and passes it to the rest of the compiler.
- Checking and translation can be done during parsing, so the parse tree need not be constructed explicitly.
- The parser and the front end can be implemented as a single module.

Types of Parsers:
1. Universal Parsers:
   - Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar.
   - Too inefficient for production compilers.

2. Top-Down Parsers:
   - Build parse trees from the root to the leaves.
   - Input is scanned from left to right, one symbol at a time.

3. Bottom-Up Parsers:
   - Build parse trees from the leaves to the root.
   - Input is scanned from left to right, one symbol at a time.

Efficient Parsing Methods:
- Efficient top-down and bottom-up methods work only for subclasses of grammars.
- LL (top-down) and LR (bottom-up) grammars are expressive enough for most syntactic constructs.
- Hand-implemented parsers often use LL grammars (e.g., predictive parsing).
- Parsers for LR grammars are usually constructed using automated tools.

Parser Output:
- Assumes the output is a representation of the parse tree for the token stream.
- During parsing, tasks like collecting token information into the symbol table, type checking, semantic analysis, and generating intermediate code are performed.
- These activities are part of the "rest of the front end."

Representative Grammars

- Focus on grammars used for parsing constructs.
- Keywords like `while` or `int` make parsing easier by guiding grammar production choice.
- Concentration on expressions due to the complexity of operator associativity and precedence.

Associativity and Precedence:
- Grammar for Expressions:
  - E represents expressions with terms separated by +.
  - T represents terms with factors separated by *.
  - F represents factors that are either parenthesized expressions or identifiers.

Example Grammar for Bottom-Up Parsing:
- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

- Suitable for bottom-up parsing (LR grammars).
- Adaptable for additional operators and precedence levels.
- Not suitable for top-down parsing due to left recursion.

Non-Left-Recursive Grammar for Top-Down Parsing:
- $E \rightarrow TE'$
- $E \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid id$

Ambiguity Handling:
- Grammar treating + and * alike:
  - $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- Useful for illustrating parsing ambiguities.
- Allows multiple parse trees for expressions like a + b * c.

Syntax Error Handling
- Discusses the nature of syntactic errors and error recovery strategies.
- Focus on panic-mode and phrase-level recovery.

Importance of Error Handling:
- Compilers need to help locate and fix errors in programs.
- Most programming languages don't specify error handling, so it's up to the compiler designer.
- Proper error handling simplifies compiler design and improves error management.

Types of Errors:
1. Lexical Errors:
   - Misspellings of identifiers, keywords, or operators.
   - Missing quotes around text strings.

2. Syntactic Errors:
   - Misplaced semicolons or braces.
   - Incorrect use of case statements in languages like C or Java.

3. Semantic Errors:
   - Type mismatches between operators and operands.

4. Logical Errors:
   - Incorrect reasoning or using the wrong operator (e.g., = instead of ==).

Detecting Errors:
- Parsing methods like LL and LR detect errors efficiently.
- Viable-prefix property: Errors detected as soon as an uncompletable input prefix is encountered.

Error Handling Goals:
1. Report errors clearly and accurately.
2. Recover quickly to detect more errors.
3. Minimize impact on processing correct programs.

Error Reporting:
- Error handler should report where an error is detected.
- Common strategy: Print the offending line with a pointer to the error position.

Parser in Compiler
Role of the Parser:
- The parser receives a string of tokens from the lexical analyzer.
- Verifies if the token string can be generated by the grammar for the source language.
- Reports syntax errors clearly and recovers from common errors to continue processing the program.

Parse Tree:
- Constructs a parse tree for well-formed programs and passes it to the rest of the compiler.
- Checking and translation can be done during parsing, so the parse tree need not be constructed explicitly.
- The parser and the front end can be implemented as a single module.

Types of Parsers:
1. Universal Parsers:
   - Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar.
   - Too inefficient for production compilers.

2. Top-Down Parsers:
   - Build parse trees from the root to the leaves.
   - Input is scanned from left to right, one symbol at a time.

3. Bottom-Up Parsers:
   - Build parse trees from the leaves to the root.
   - Input is scanned from left to right, one symbol at a time.

Efficient Parsing Methods:
- Efficient top-down and bottom-up methods work only for subclasses of grammars.
- LL (top-down) and LR (bottom-up) grammars are expressive enough for most syntactic constructs.
- Hand-implemented parsers often use LL grammars (e.g., predictive parsing).
- Parsers for LR grammars are usually constructed using automated tools.

Parser Output:
- Assumes the output is a representation of the parse tree for the token stream.
- During parsing, tasks like collecting token information into the symbol table, type checking, semantic analysis, and generating intermediate code are performed.
- These activities are part of the "rest of the front end."

Context-Free Grammars

Introduction:
- Context-free grammars systematically describe the syntax of programming language constructs like expressions and statements.
- Syntax variables, such as `stmt` for statements and `expr` for expressions, are used in these grammars.

Example Production:
- The production `stmt → if ( expr ) stmt else stmt` specifies the structure of a conditional statement.
- Other productions define what `expr` and other forms of `stmt` can be.

Purpose of this Section:
- Reviews the definition of context-free grammar.
- Introduces terminology for discussing parsing.
- Highlights the concept of derivations, which is useful for understanding the order in which productions are applied during parsing.

The Formal Definition of a Context-Free Grammar

A context-free grammar consists of terminals, nonterminals, a start symbol, and productions.

1. Terminals:
   - Basic symbols used to form strings.
   - Synonym: "token name" or "token."
   - Example: In the production `stmt → if ( expr ) stmt else stmt, the terminals are `if`, `else`, `(`, and `)`.

2. Nonterminals:
   - Syntactic variables representing sets of strings.
   - Example: `stmt` and `expr` in the production above.
   - Nonterminals define the language's hierarchical structure.

3. Start Symbol:
   - A designated nonterminal that represents the language generated by the grammar.
   - Productions for the start symbol are typically listed first.

4. Productions:
   - Rules that specify how terminals and nonterminals combine to form strings.
   - Each production has:
     - Head (left side): A nonterminal defining some of its strings.
     - Arrow symbol (`→`).
     - Body (right side): A sequence of terminals and nonterminals describing string construction.

Simple Arithmetic Expressions Grammar

Grammar Overview:
- Defines simple arithmetic expressions.
- Terminal Symbols: id, +, -, *, (,)
- Nonterminal Symbols: expression, term, factor
- Start Symbol: expression

Productions:
1. Expression:
   - expression → expression + term
   - expression → expression - term
   - expression → term

2. Term:
   - term → term * factor
   - term → term / factor
   - term → factor

3. Factor:
   - factor → (expression)
   - factor → id

This grammar specifies how arithmetic expressions are structured using terminals and nonterminals. The start symbol `expression` can be expanded into more complex structures by applying the productions.

Notational Conventions

Purpose: To avoid repeatedly defining terminals, nonterminals, and other elements in grammars, specific notational conventions are used.

Notational Conventions:

1. Terminals:
   - Lowercase letters early in the alphabet: a, b, c, etc.
   - Operator symbols: +, *, etc.
   - Punctuation symbols: parentheses, commas, etc.
   - Digits: 0-9
   - Boldface strings: id, if

2. Nonterminals:
   - Uppercase letters early in the alphabet: A, B, C
   - The letter S (usually the start symbol)
   - Lowercase, italic names: expr, stmt
   - Uppercase letters for constructs: E (expressions), T (terms), F (factors)

3. Grammar Symbols:
   - Uppercase letters late in the alphabet: X, Y, Z (represent both terminals and nonterminals)

4. Strings of Terminals:
   - Lowercase letters late in the alphabet: u, v, w, x (represent strings of terminals)

5. Strings of Grammar Symbols:
   - Lowercase Greek letters: α (alpha), β (beta), γ (gamma) (represent strings of grammar symbols)

6. Productions with Common Head:
   - Written as: `A → α1 / α2 / ... / αk`
   - α1, α2,..., αk are alternatives for A.

7. Start Symbol: The head of the first production is the start symbol (unless stated otherwise).

Example of Conventions:
- Grammar:
  - E → E + T | E - T | T
  - T → T * F | T / F | F
  - F → (E) | id
- Explanation:
  - E, T, F are nonterminals (with E as the start symbol).
  - Symbols like +, -, *, /, (,), and id are terminals.

Derivations

Overview:
- Derivations treat grammar productions as rewriting rules to construct a parse tree.
- Each rewriting step replaces a nonterminal with the body of one of its productions.
- Useful for understanding both top-down and bottom-up parsing.

Types of Derivations:
1. Leftmost Derivations:
   - Replace the leftmost nonterminal in each step.
   - Example: E → -E → -(E) → -(id)

2. Rightmost Derivations:
   - Replace the rightmost nonterminal in each step.
   - Related to bottom-up parsing.
   - Example: E → -E → -(E) → -(E + E) → -(E + id) → -(id + id)

General Definition of Derivation:
- For a nonterminal A in a sequence of grammar symbols (e.g., aAβ):
  - If A → γ is a production, we write aAβ ⇒ aγβ (derives in one step).
  - Sequence of derivations: a1 ⇒ a2 ⇒ ... ⇒ an (derives in multiple steps).
  - α ⇒* β: Derives in zero or more steps.
  - α ⇒+ β: Derives in one or more steps.

Sentential Forms and Sentences:
- Sentential Form: A string with terminals and nonterminals derived from the start symbol.
- Sentence: A sentential form with only terminals.
- Language: The set of sentences generated by a grammar.

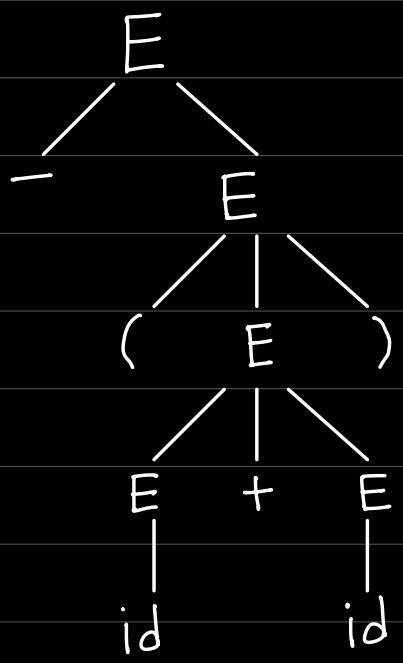
Example Grammar:
- Grammar: $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$
- Derivation Example:
  - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$
  - (id + id) is a sentence of the grammar.

Leftmost and Rightmost Derivations:
- Leftmost Step: $wAy \Rightarrow_{lm} w\gamma y$ (where w is terminals only, $A \rightarrow \gamma$ is applied)
- Rightmost Step: Similar definition for rightmost derivations (canonical derivations).

Parse Trees and Derivations

Parse Trees:
- A graphical representation of a derivation.
- Interior nodes represent the application of a production.
- Labeled with the nonterminal from the head of the production.
- Children of the node are labeled by symbols in the production body.
- Leaves are labeled by nonterminals or terminals, constituting a sentential form called the yield or frontier of the tree.

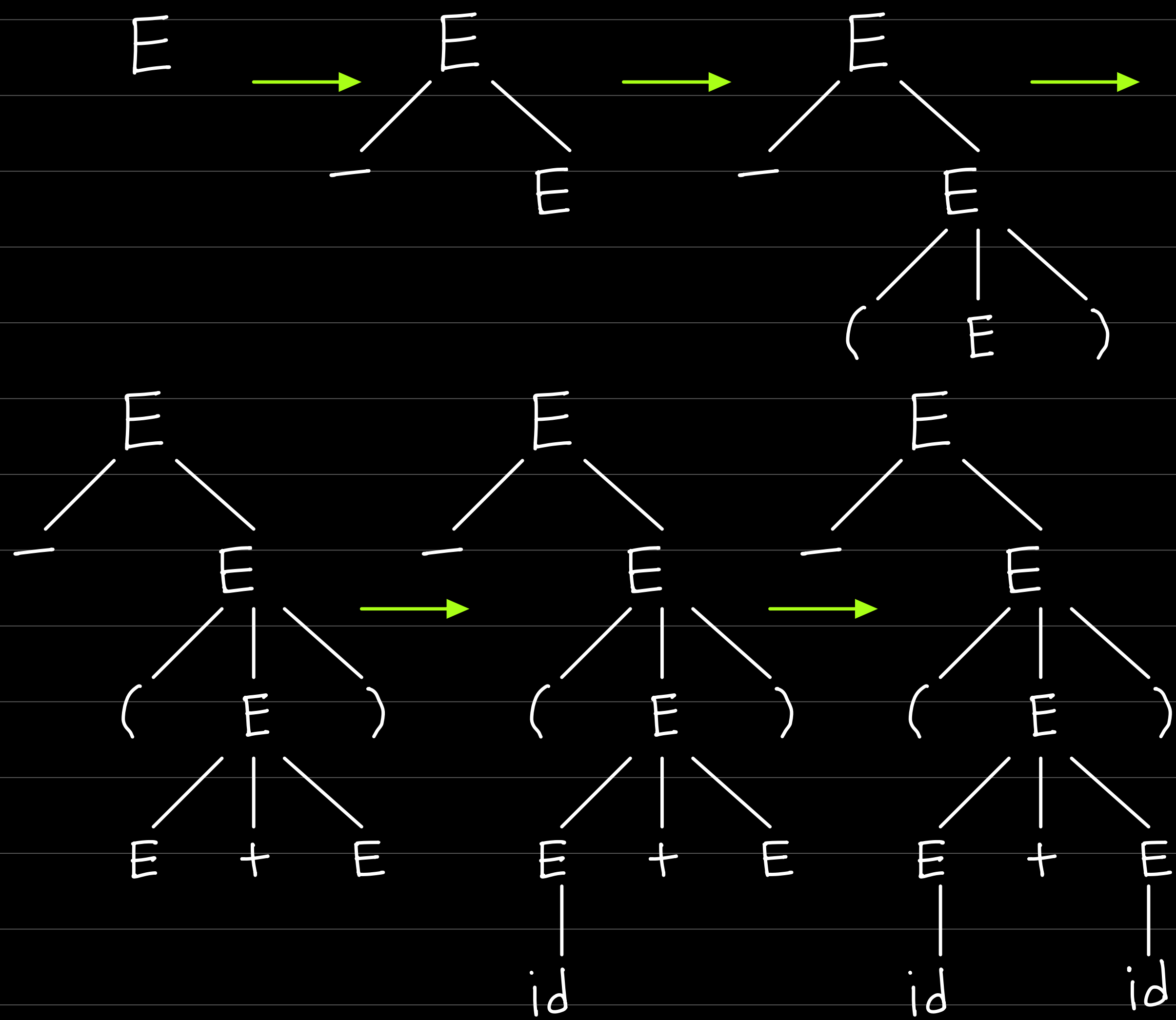
Example:
- Derivation Example for -(id + id):



Derivations:
- Leftmost Derivation: Replace the leftmost nonterminal first.
- Rightmost Derivation: Replace the rightmost nonterminal first.
- Each derivation step replaces a nonterminal with its production body.
- Leftmost and rightmost derivations help filter out variations in the order of replacements.
- Parse trees can have multiple derivations but correspond to unique leftmost and rightmost derivations.

Relationship Between Parse Trees and Derivations:**
- Both leftmost and rightmost derivations can represent the same parse tree.
- Example Derivations for -(id + id):
  - Leftmost: E → -E → -(E) → -(E + E) → -(id + E) → -(id + id)
  - Rightmost: E → -E → -(E) → -(E + E) → -(E + id) → -(id + id)

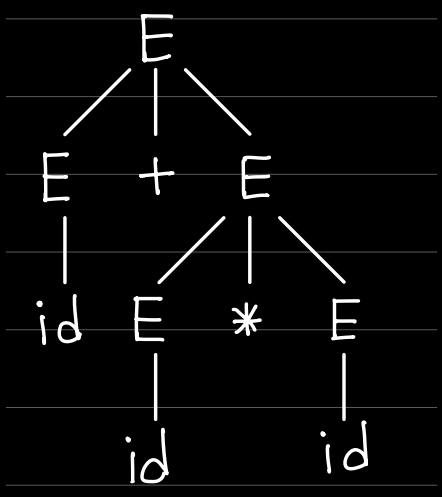
Sequence of parse tree for derivation:


E
E → – E
E → – (E)
E → – (E + E)
E → – (id + E)
E → – (id + id)

Parse Trees for Ambiguous Grammar Example

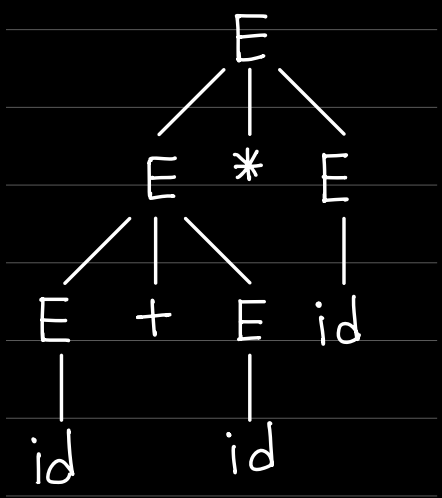
Here are the two distinct parse trees for the sentence id + id * id:

Parse Tree 1: (Reflecting operator precedence, with * having higher precedence than +)



E
—> E + E
—> id + E
—> id + E * E
—> id + id * E
—> id + id * id

Parse Tree 2: (Not reflecting operator precedence, with + being evaluated first)



E
—> E * E
—> E + E * E
—> id + E * E
—> id + id * E
—> id + id * id

Explanation:
- Parse Tree 1: Reflects the commonly assumed precedence where * has higher precedence than +. So, id + (id * id) is evaluated as id + (id * id).
- Parse Tree 2: Does not reflect precedence, treating + as having the same precedence as *. So, (id + id) * id is evaluated as (id + id) * id.

These examples illustrate how the same sentence can have multiple parse trees, making the grammar ambiguous. This ambiguity can affect the evaluation order of expressions, which is why it's often desirable to resolve ambiguity in grammars.