S J P N Trust's

HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI.

Inculcating Values, Promoting Prosperity

Approved by AICTE, Recognized by Govt. of Karnataka and Permanently Affiliated to VTU Belagavi.

Accredited at 'A' Grade by NAAC

Programmes Accredited by NBA: CSE, ECE, EEE & ME

Subject: System Software and Compilers (18CS61)

Module 2: Introduction to Compilers and Lexical Analysis

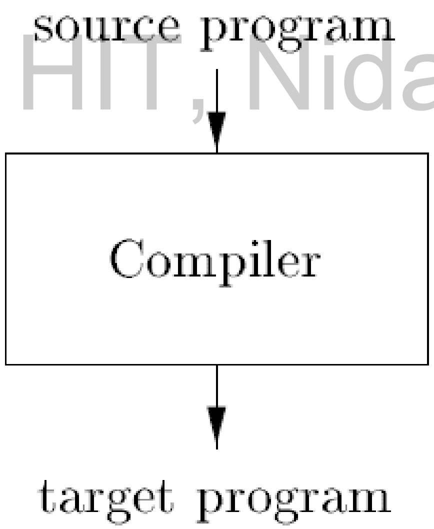
Dr. Mahesh G. Huddar

Dept. of Computer Science and Engineering

Language Processors

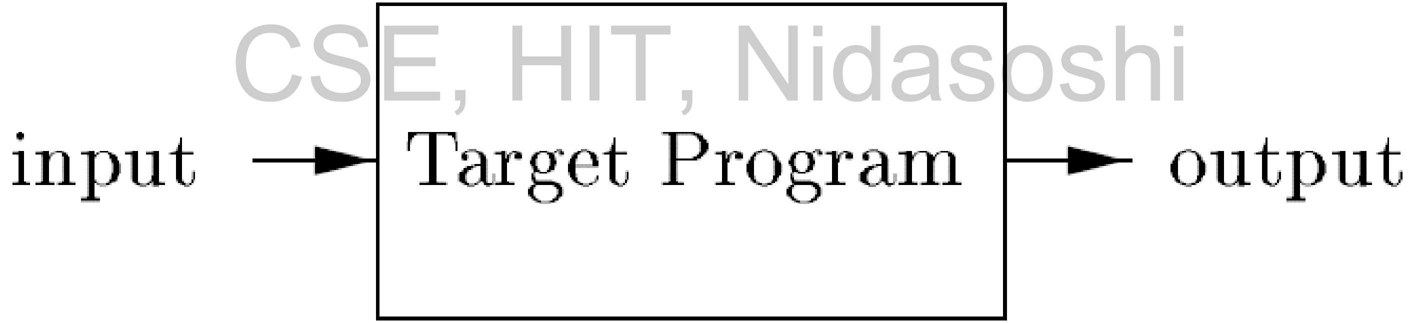
Definition:

- 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**



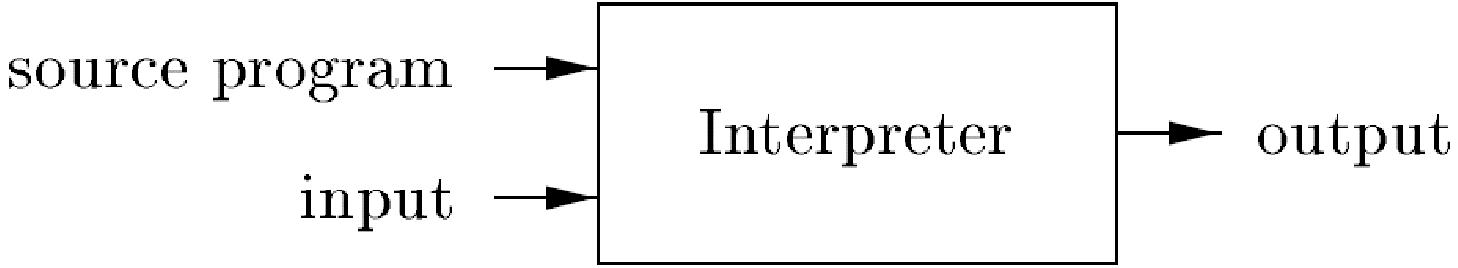
Language Processors

- If the **target program** is an **executable machine-language program**, it can then be called by the user to process inputs and produce outputs



Language Processors

- An **interpreter** is 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



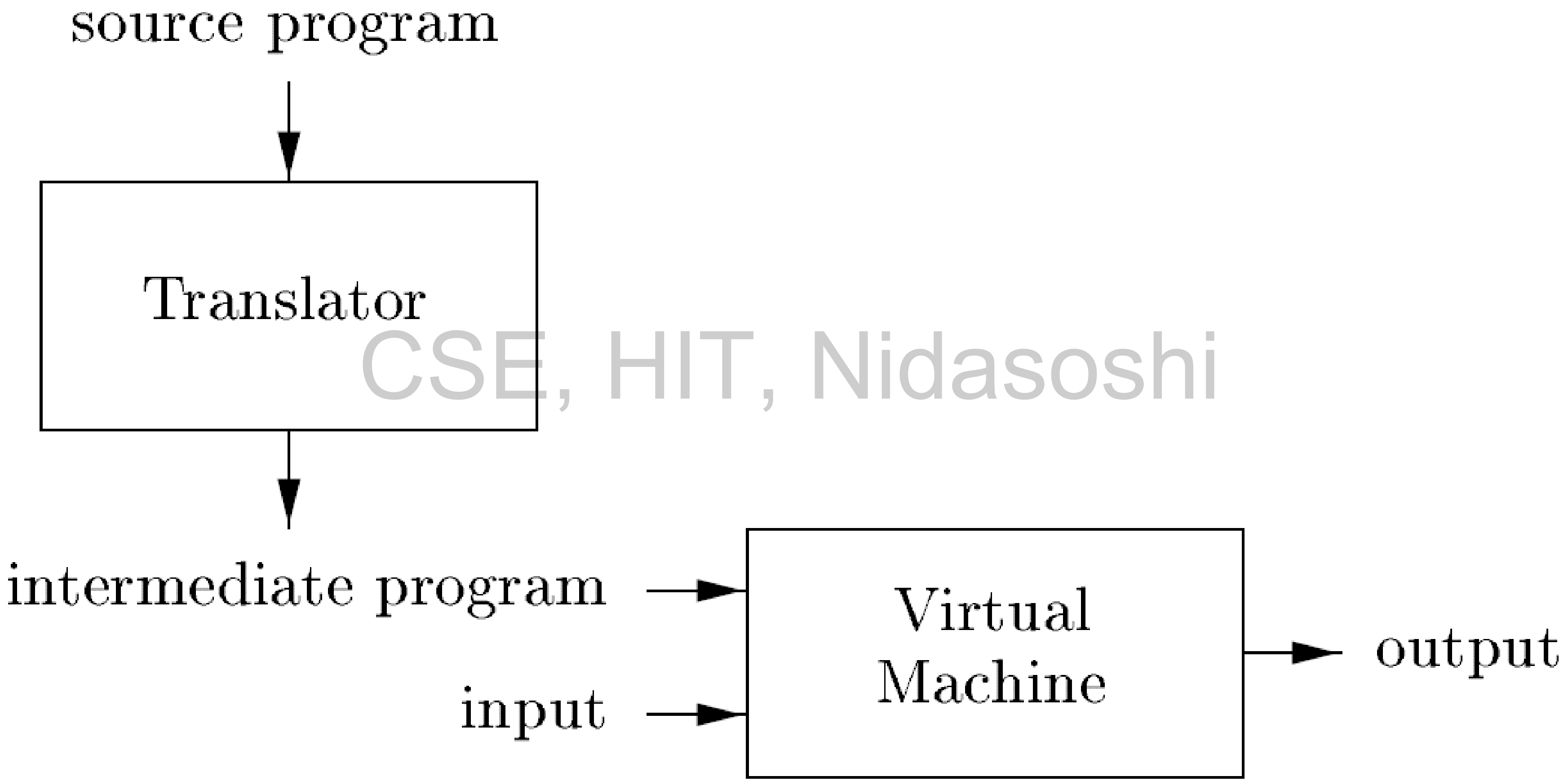
Language Processors

- The machine-language (target program) produced by a compiler is usually much **faster** than an interpreter at mapping inputs to outputs .
- An interpreter, however, can usually give **better error diagnostics** than a compiler, because it executes the source program statement by statement.

Language Processors

- Java language processors combine compilation and interpretation, as shown in Fig.
- A Java source program may first be compiled into an intermediate form called bytecodes.
- The bytecodes are then interpreted by a virtual machine.
- A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.
- In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

Language Processors



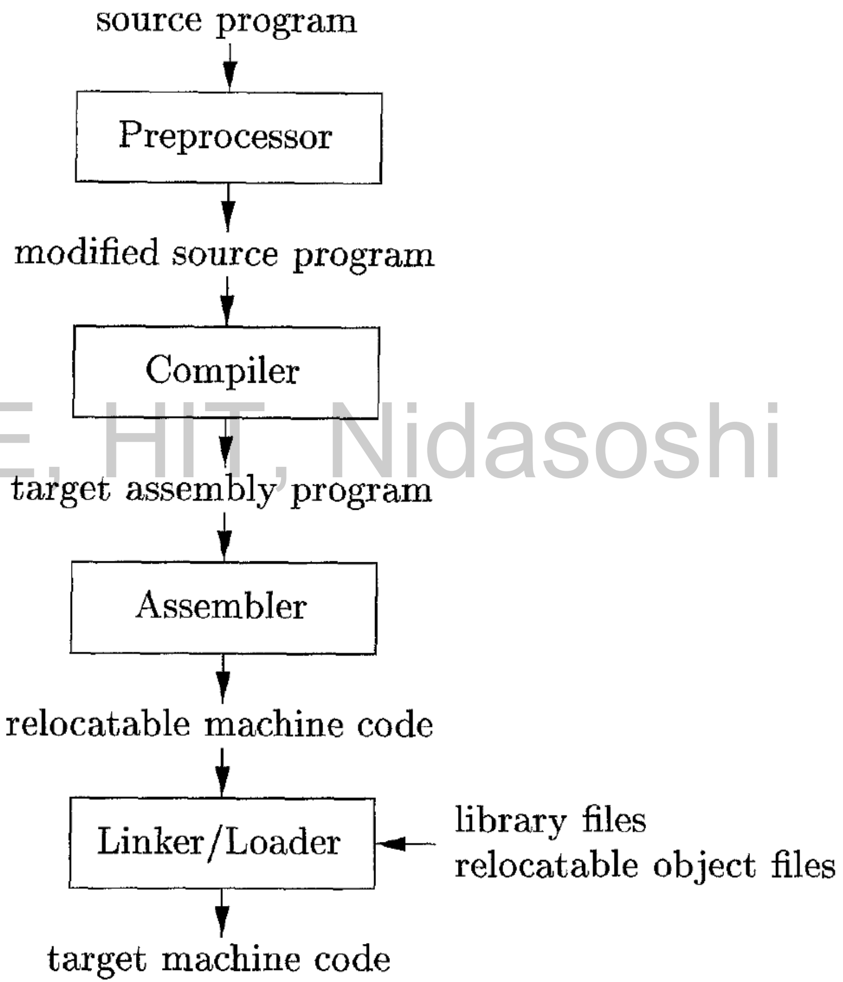
Language Processors

- In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig.
  - A source program may be divided into modules stored in separate files.
  - The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor.
  - The preprocessor may also expand shorthand's, called macros, into source language statements.
  - The modified source program is then fed to a compiler.
  - The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.

Language Processors

- The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.
- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The linker resolves external memory addresses, where the code in one file may refer to a location in another file. The loader then puts together all of the executable object files into memory for execution.

Language Processors



The Structure of a Compiler

There are two parts in compiler: **analysis** and **synthesis**.

- The **analysis part** breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.
- 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 Structure of a Compiler

There are two parts in compiler: **analysis** and **synthesis**.

- The **synthesis part** constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called the front end of the compiler; the synthesis part is the back end.

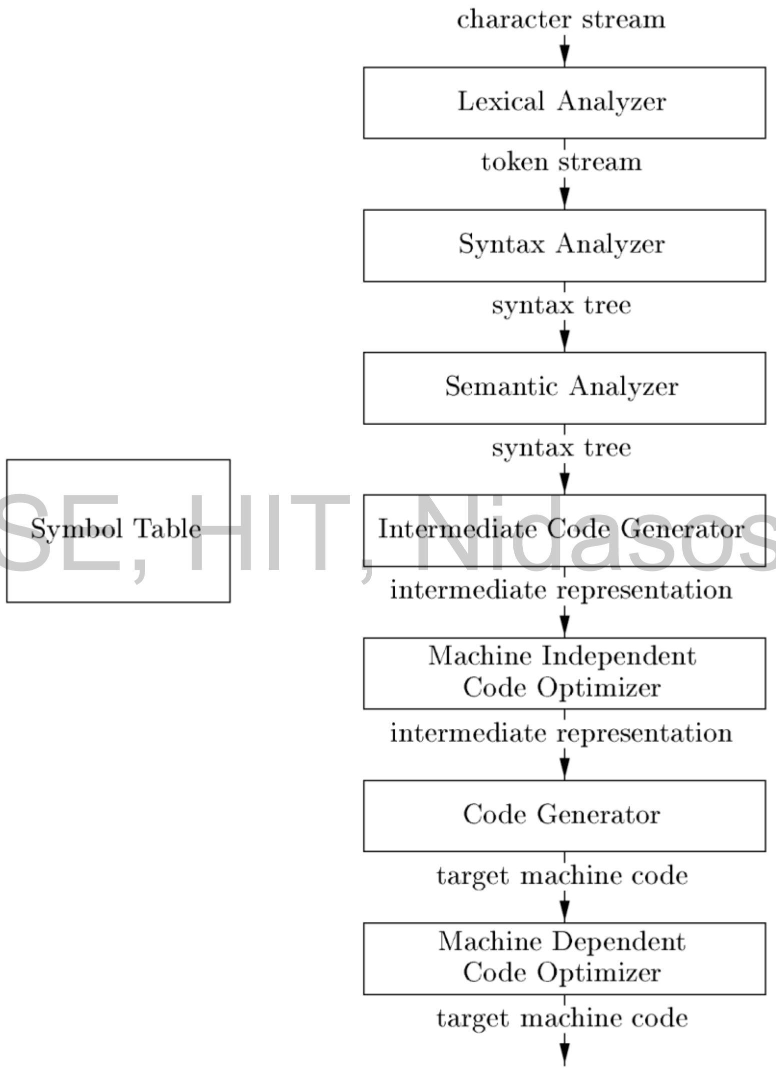
The Structure of a Compiler

- If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another.
- A typical decomposition of a compiler into phases is shown in Fig. (Refer next slide)
- In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly.
- The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

The Structure of a Compiler

- Some compilers have a machine-independent optimization phase between the front end and the back end.
- The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation.
- Since optimization is optional, one or the other of the two optimization phases shown in Fig. maybe missing.

The Structure (Phases) of a Compiler



Lexical Analysis

- The first phase of a compiler is called **lexical analysis or 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 lexeme, 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.
- In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.
- Information from the symbol-table entry is needed for semantic analysis and code generation.

Lexical Analysis

- For example, suppose a source program contains the assignment statement

position = initial + rate * 60 (1.1)

- The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. position is a lexeme that would be mapped into a token (id, 1), where id is an abstract symbol standing for identifier and 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.

Lexical Analysis

2. The assignment symbol = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as assign for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.
3. initial is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial .
4. + is a lexeme that is mapped into the token (+).
5. rate is a lexeme that is mapped into the token (id, 3), where 3 points to the symbol-table entry for rate .
6. * is a lexeme that is mapped into the token (*) .
7. 60 is a lexeme that is mapped into the token (60)

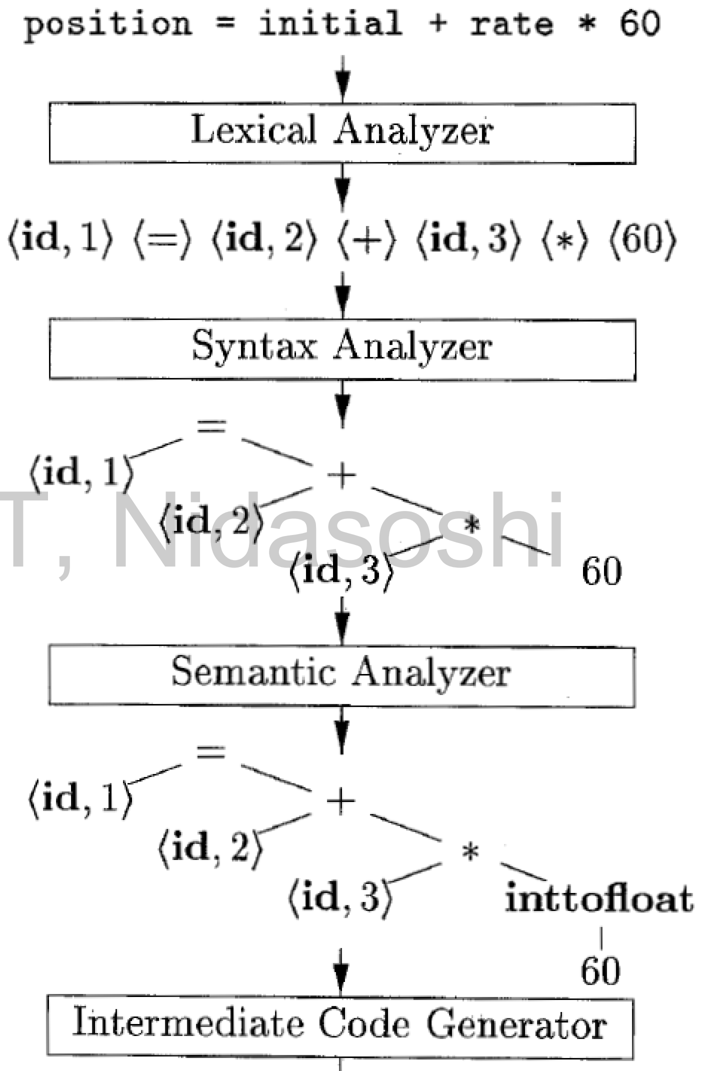
1. Lexical Analysis

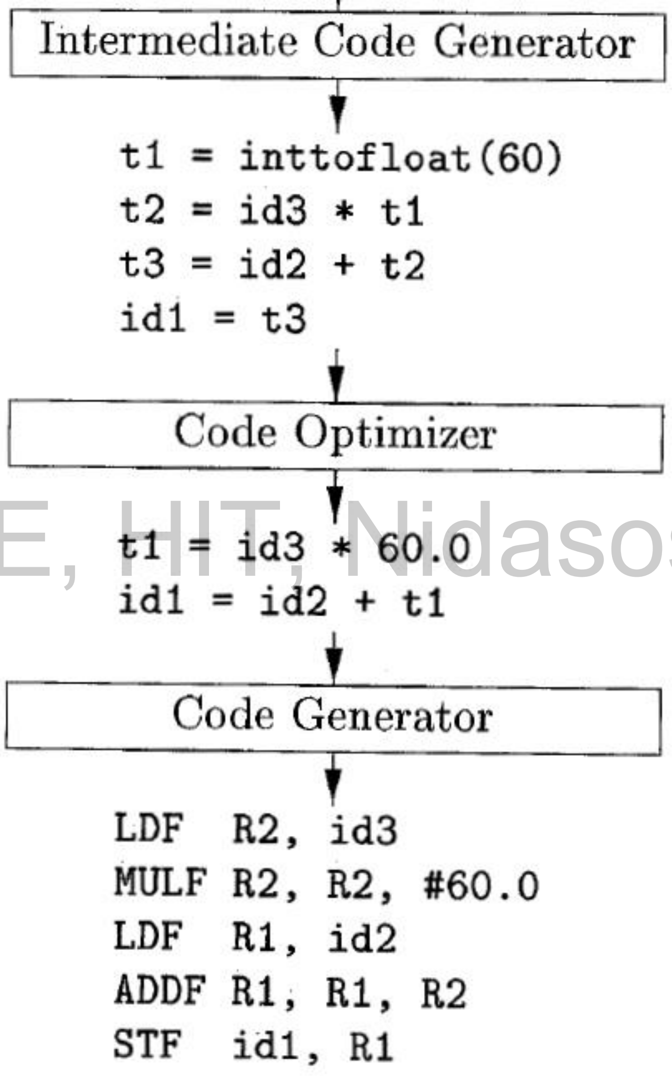
- Figure shows the representation of the assignment statement after lexical analysis as the sequence of tokens

$$\langle \mathbf{id}, 1\rangle \; \langle = \rangle \; \langle \mathbf{id}, 2\rangle \; \langle + \rangle \; \langle \mathbf{id}, 3\rangle \; \langle * \rangle \; \langle 60 \rangle \qquad \textbf{(1.2)}$$

1	position	⋯
2	initial	⋯
3	rate	⋯

SYMBOL TABLE





```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

2. Syntax Analysis

- The second phase of the compiler is syntax analysis or parsing.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.
- A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig.
- This tree shows the order in which the operations in the assignment

position = initial + rate * 60

- are to be performed.

2. Syntax Analysis

- The tree has an interior node labeled * with (id, 3) as its left child and the integer 60 as its right child.
- The node (id, 3) represents the identifier rate.
- The node labeled * makes it explicit that we must first multiply the value of **rate** by 60.
- The node labeled + indicates that we must add the result of this multiplication to the value of initial .
- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier posit ion.
- This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

3. Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.
- For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

3. Semantic Analysis

- The language specification may permit some type conversions called coercions.
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers.
- If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.
- Suppose that **position**, **initial** , and **rate** have been declared to be floating-point numbers, and that the lexeme 60 by itself forms an integer.
- The type checker in the semantic analyzer in Fig. discovers that the operator * is applied to a floating-point number **rate** and an integer 60.
- In this case, the integer may be converted into a floating-point number.

4. Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.
- Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.
- many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.
- This intermediate representation should have two important properties: it should be easy to produce, and it should be easy to translate into the target machine.

4. Intermediate Code Generation

- The output of the intermediate code generator in,

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

(1.3)

5. Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.
- The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.
- Moreover, t3 is used only once to transmit its value to id1 so the optimizer can transform (1.3) into the shorter sequence

```
t1 = id3 * 60.0
id1 = id2 + t1
```

(1.4)

Design Objectives of Compiler optimizations

1. The optimization must be correct, that is, preserve the meaning of the compiled program,
2. The optimization must improve the performance of many programs,
3. The compilation time must be kept reasonable, and
4. The engineering effort required must be manageable.

6. Code Generation

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
- A crucial aspect of code generation is the judicious assignment of registers to hold variables.

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2                    (1.5)
ADDF R1,  R1, R2
STF  id1, R1
```

6. Code Generation

- The first operand of each instruction specifies a destination.
- The F in each instruction tells us that it deals with floating-point numbers.
- The code in (1.5) loads the contents of address id3 into register R2, then multiplies it with floating-point constant 60.0.

- The # signifies that 60.0 is to be treated as an immediate constant. The third instruction moves id2 into register R1 and the fourth adds to it the value previously computed in register R2.
- Finally, the value in register R1 is stored into the address of idl, so the code correctly implements the assignment statement (1.1).

Symbol-Table Management

- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.
- These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.
- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

The Grouping of Phases into Passes

- The discussion of phases deals with the logical organization of a compiler.
- In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
- For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
- Code optimization might be an optional pass.
- Then there could be a back-end pass consisting of code generation for a particular target machine

Compiler-Construction Tool

- The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on.
- In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.
- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

Compiler-Construction Tool

Some commonly used compiler-construction tools include

1. **Parser generators** that automatically produce syntax analyzers from a grammatical description of a programming language.
2. **Scanner generators** that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. **Syntax-directed translation** engines that produce collections of routines for walking a parse tree and generating intermediate code.
4. **Code-generator generators** that produce a code from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. **Data-flow analysis engines** that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.
6. **Compiler-construction toolkits** that provide an integrated set of routines for constructing various phases of a compiler

The Evolution of Programming Languages

- The first electronic computers appeared in the 1940's and were programmed in **machine language by sequences of 0's and 1's** that explicitly told the computer what operations to execute and in what order.

- The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on.
- Needless to say, this kind of programming was slow, tedious, and error prone. And once written, the programs were hard to understand and modify.

The Move to Higher-level Languages

- The first step towards more people-friendly programming languages was the development of **mnemonic assembly languages** in the early 1950’s.
- Initially, the instructions in an assembly language were just mnemonic representations of machine instructions.
- Later, **macro instructions were added** to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

The Move to Higher-level Languages

- Today, there are thousands of programming languages.
- They can be classified in a variety of ways.
- **One classification is by generation.**
  - **First-generation** languages are the machine languages,
  - **Second-generation** the assembly languages,
  - **Third-generation** the higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java.
  - **Fourth-generation** languages are languages designed for specific applications like SQL for database queries, and Postscript for text formatting.
  - The term **fifth-generation** language has been applied to logic- and constraint-based languages like Prolog and OPS5.

The Move to Higher-level Languages

- **Another classification of languages uses the term imperative for languages** in which a program specifies how a computation is to be done and declarative for languages in which a program specifies what computation is to be done.

- Languages such as C, C++, C#, and Java are imperative languages.

The Move to Higher-level Languages

- The **term von Neumann language** is applied to programming languages whose computational model is based on the von Neumann computer architecture.
- Many of today's languages, such as Fortran and C are von Neumann languages.
- An **object-oriented language** is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another.
- Simula 67 and Smalltalk are the earliest major object-oriented languages.
- Languages such as C++, C#, Java, and Ruby are more recent object-oriented languages.

The Move to Higher-level Languages

- **Scripting languages** are interpreted languages with high-level operators designed for "gluing together" computations.
- These computations were originally called "scripts." Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl are popular examples of scripting languages.
- Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

Impacts on Compilers

- Since the design of programming languages and compilers are intimately related, the advances in programming languages placed new demands on compiler writers.
- They had to devise algorithms and representations to translate and support the new language features. Since the 1940's, computer architecture has evolved as well.
- Not only did the compiler writers have to track new language features, they also had to devise translation algorithms that would take maximal advantage of the new hardware capabilities.

Impacts on Compilers

- Compiler writing is challenging. A compiler by itself is a large program.
- Moreover, many modern language-processing systems handle several source languages and target machines within the same framework; that is, they serve as collections of compilers, possibly consisting of millions of lines of code.
- Consequently, good software-engineering techniques are essential for creating and evolving modern language processors.
- A compiler must translate correctly the potentially infinite set of programs that could be written in the source language.
- The problem of generating the optimal target code from a source program is undecidable in general; thus, compiler writers must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.

Applications of Compiler Technology

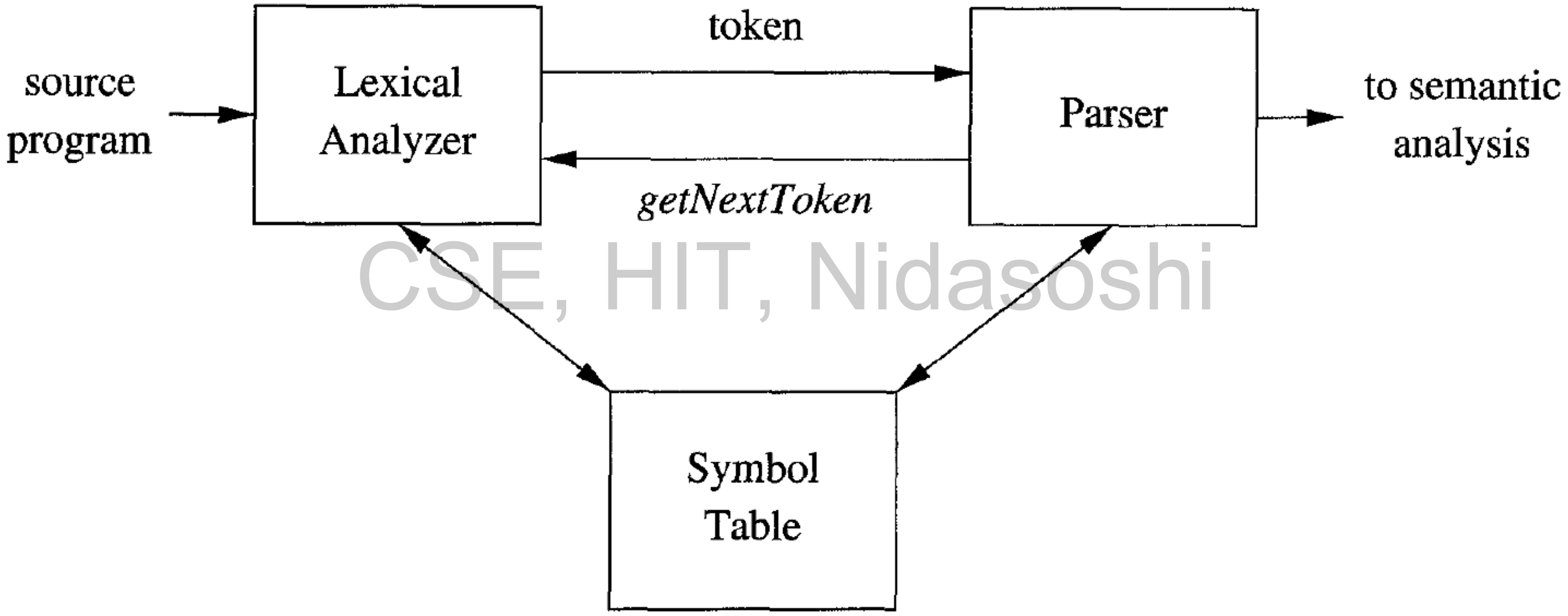
Lexical Analysis – The Role of the Lexical Analyzer

- As the first phase of a compiler, the **main task** of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.
- The stream of tokens is sent to the **parser** for syntax analysis.
- It is common for the lexical analyzer to interact with the **symbol table** as well.

Lexical Analysis – The Role of the Lexical Analyzer

- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- Commonly, the interaction lexical analyzer and syntax analyzer is implemented by having the **parser call** the lexical analyzer.
- The call, suggested by the **getNextToken** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

Lexical Analysis – The Role of the Lexical Analyzer



Lexical Analysis – The Role of the Lexical Analyzer

- Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain **other tasks** besides identification of lexemes.
- One such task is **stripping out comments and whitespace** (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Another task is **correlating error messages** generated by the compiler with the source program.
  - For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
- If the source program uses a macro-preprocessor, the **expansion of macros** may also be performed by the lexical analyzer.

Lexical Analysis – The Role of the Lexical Analyzer

- Sometimes, lexical analyzers are divided into a cascade of two processes:

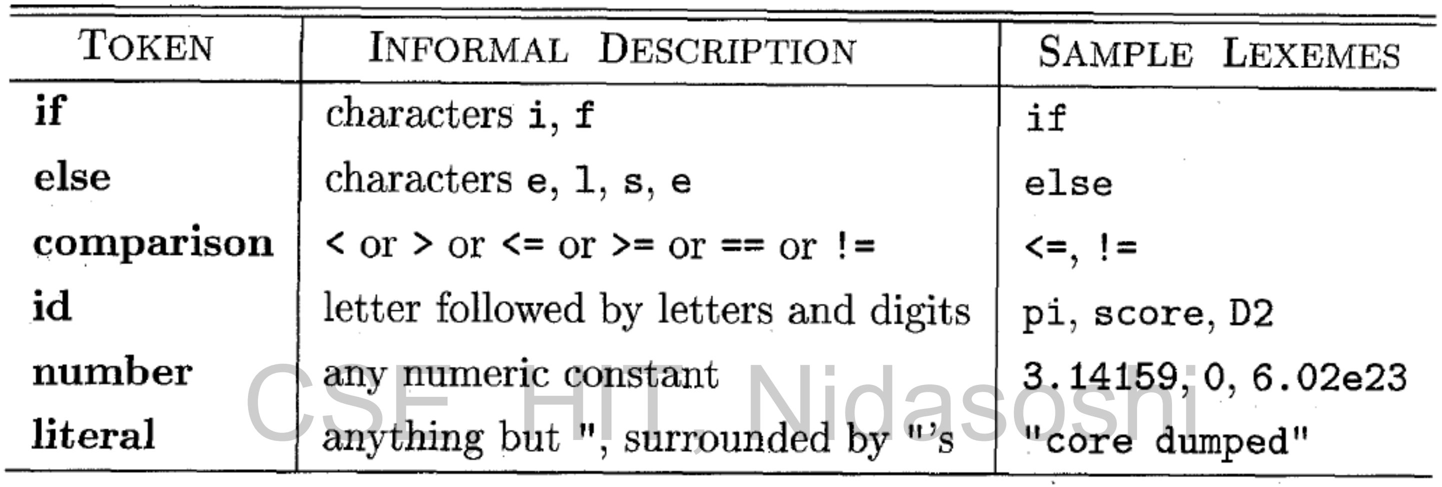
  a) **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.

  b) **Lexical analysis** is more complex portion, where the scanner produces the sequence of tokens as output.

Tokens, Patterns, and Lexemes

1. A **token** is a pair consisting of a token name and an optional attribute value.
   - The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier.
   - The token names are the input symbols that the parser processes.
2. A **pattern** is a description of the form that the lexemes of a token may take.
   - In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
   - For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.
3. A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Tokens, Patterns, and Lexemes



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

- Example: Figure gives some typical tokens, their informally described patterns, and some sample lexemes. In the C statement

  printf ("Total = %d\n", score) ;

- both **printf** and **score** are lexemes matching the pattern for **keyword, id**, and **"Total = %d\n"** is a lexeme matching **literal**.

Tokens, Patterns, and Lexemes

In many programming languages, the following classes cover **most or all the tokens**:

1. One token for each **keyword**.
   - The pattern for a keyword is the same as the keyword itself
2. Tokens for the **operators**, either individually or in classes such as the token **Comparison**

3. One token representing all **identifiers**.
4. One or more tokens representing **constants**, such as numbers and literal strings.
5. Tokens for each **punctuation symbol**, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens

The token names and associated attribute values for the Fortran statement

E = M * C ** 2

are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>

< assign-op >

<id, pointer to symbol-table entry for M>

<mult -op>

<id, pointer to symbol-table entry for C>

<exp-op>

<number , integer value 2 >

Note that in certain pairs, especially **operators**, **punctuation**, and **keywords**, there is no need for an attribute value. In this example, the token **number** has been given an **integer-valued** attribute.

In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for **number** a pointer to that string

Attributes for Tokens - Example

- **Divide the following C + + program:**

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

- **Solution:**

```
<float> <id, limitedSquaare> <(> <id, x> <)>
{> <float> <id, x>
<return> <(> <id, x> <op,"<="> <num, -10.0> <op, "||"> <id, x>
<op, ">="> <num, 10.0> <)> <op, "?"> <num, 100> <op, ":">
<id, x> <op, "*"> <id, x>
<}>
```

Lexical Errors

- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string **fi** is encountered for the first time in a C program in the context:

```
fi ( a == f(x)) ...
```

- a lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or an undeclared function identifier.
- Since **fi** is a **valid lexeme** for the **token id**, the lexical analyzer must return the token id to the parser and let some other phase of the compiler - probably the parser in this case - handle an error due to transposition of the letters.

Lexical Errors

- However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.

- The simplest recovery strategy is **"panic mode" recovery**. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate.

Lexical Errors

- Other possible error-recovery actions are:

  1. Delete one character from the remaining input.

  2. Insert a missing character into the remaining input.

  3. Replace a character by another character.

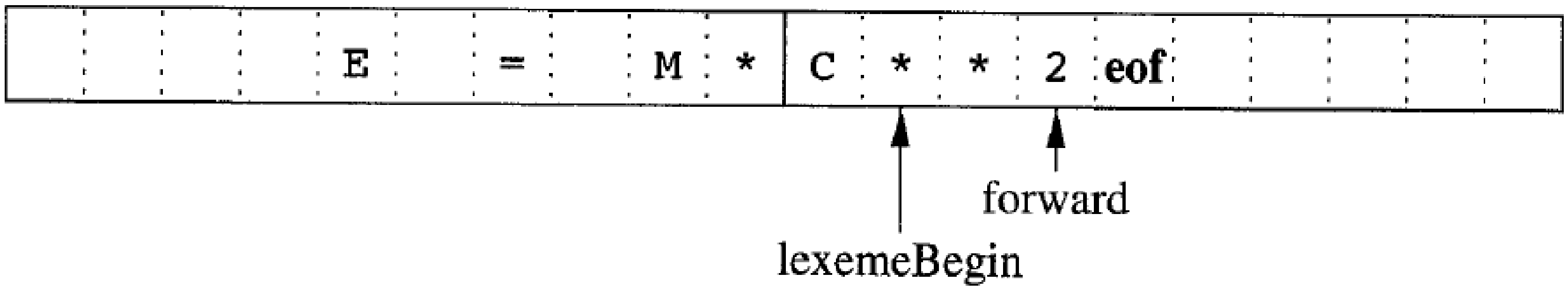
  4. Transpose two adjacent characters.

Input Buffering

- Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded.
- This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme

Input Buffering - Buffer Pairs

- Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.



Input Buffering - Buffer Pairs

- Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes.
- Using one system read command we can read N characters into a buffer, rather than using one system call per character.
- If fewer than N characters remain in the input file, then a special character, represented by **eof** marks the end of the source file and is different from any possible character of the source program.

Input Buffering - Buffer Pairs

Two pointers to the input are maintained:

1. Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer **forward** scans ahead until a pattern match is found

Input Buffering - Buffer Pairs

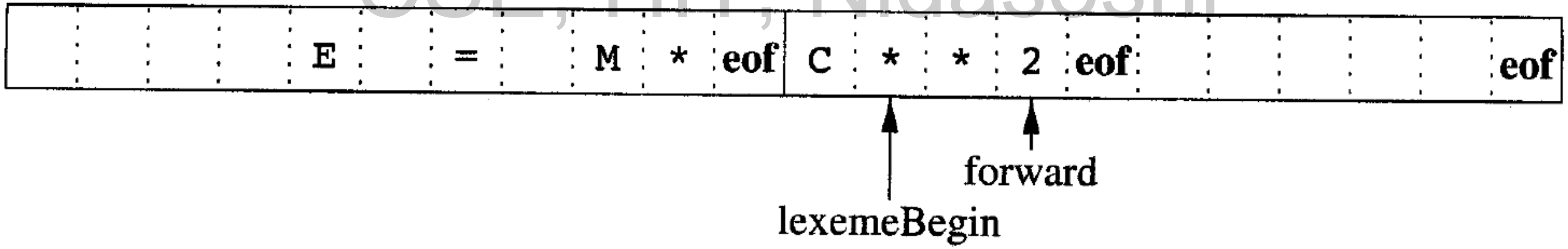
- Once the next lexeme is determined, forward is set to the character at its right end.
- Then, after the lexeme is recorded as an attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found.

Input Buffering - Sentinels

- If we use the scheme, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer.
- Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read.
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

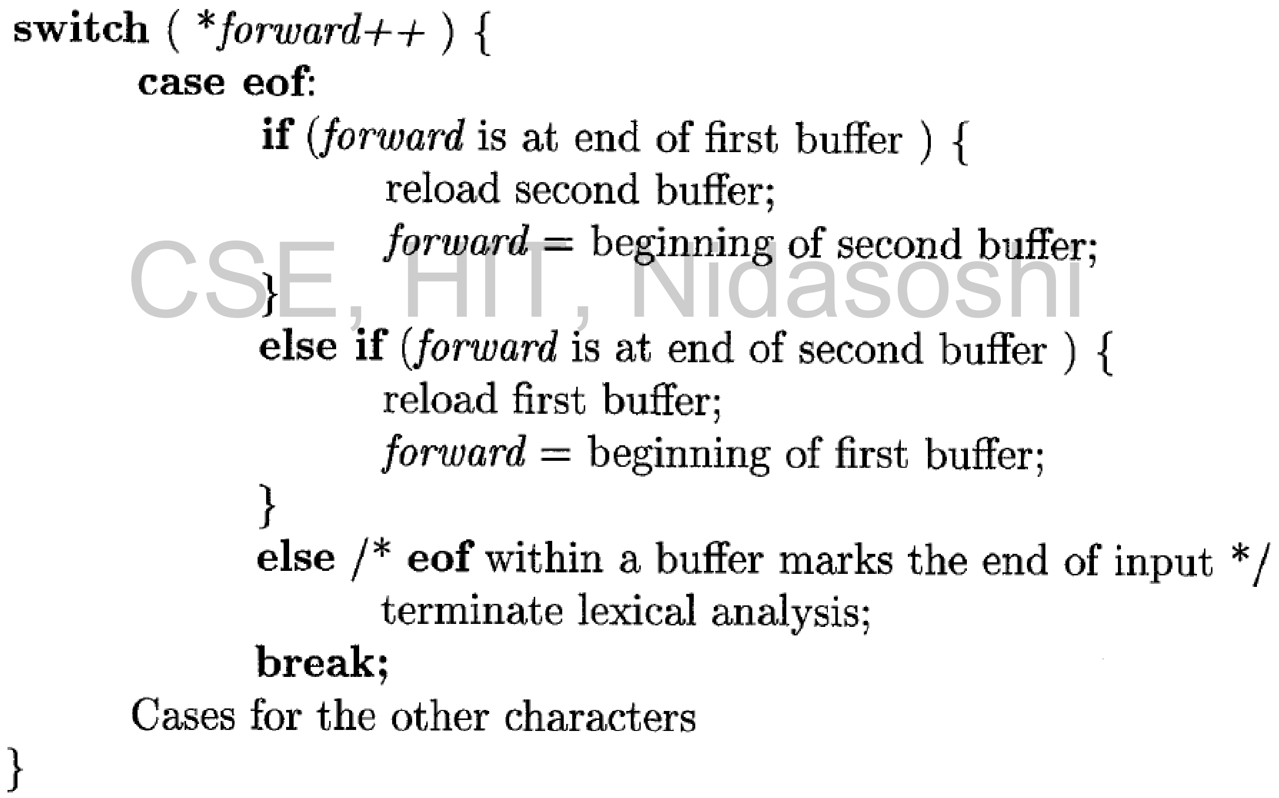
Input Buffering - Sentinels

- Figure shows the use of sentinels added.
- Note that **eof** retains its use as a marker for the end of the entire input.
- Any **eof** that appears other than at the end of a buffer means that the input is at an end.



Input Buffering - Sentinels

- Figure summarizes the algorithm for advancing forward.



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

Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. **Simplicity** of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
2. Compiler **efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. **Compiler portability is enhanced**. Input-device-specific peculiarities can be restricted to the lexical analyzer.

Specification of Tokens

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

Specification of Tokens - Strings and Languages

- An **alphabet** is any finite set of symbols.
- Typical examples of symbols are **letters, digits, and punctuation**.
- The set {0,1} is the binary alphabet.
- ASCII is an important example of an alphabet; it is used in many software systems.
- Unicode, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

Specification of Tokens - Strings and Languages

- A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
- In language theory, the terms **"sentence"** and **"word"** are often used as synonyms for "string."
- The **length** of a string s, usually written **|s|**, is the number of occurrences of symbols in s.
- For example, banana is a string of length six.
- The empty string, denoted €, is the string of length zero

Specification of Tokens - Strings and Languages

- A language is any countable set of strings over some fixed alphabet.
- This definition is very broad.
- Abstract languages like ɸ, the empty set, or (€), the set containing only the empty string, are languages under this definition.

Specification of Tokens - Strings and Languages

- If x and y are strings, then the concatenation of x and y, denoted xy, is the string formed by appending y to x.
- For example, if x = dog and y = house, then xy = doghouse.
- The empty string is the identity under concatenation; that is, for any string s, €S = S€ = s.

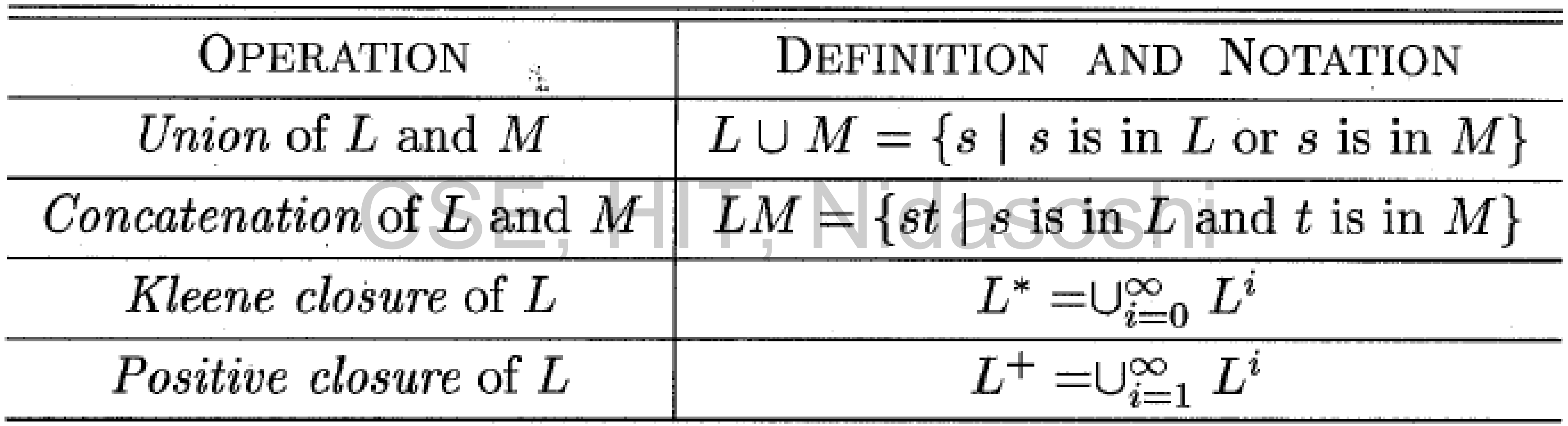
Specification of Tokens - Strings and Languages

- The **“exponentiation”** of strings is defined as follows.
- Define s^o to be €, and for all $i > 0$, define s^i to be $s^{i-1}s$.
- Since € S = S, it follows that $s^1 = s$. Then $s^2 = ss$, $s^3 = sss$, and so on.

Specification of Tokens - Operations on Languages

- In lexical analysis, the most important operations on languages are union, concatenation, and closure.
- 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.

Specification of Tokens - Operations on Languages



OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Specification of Tokens - Operations on Languages

Example: 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 may think of L and D in many ways.

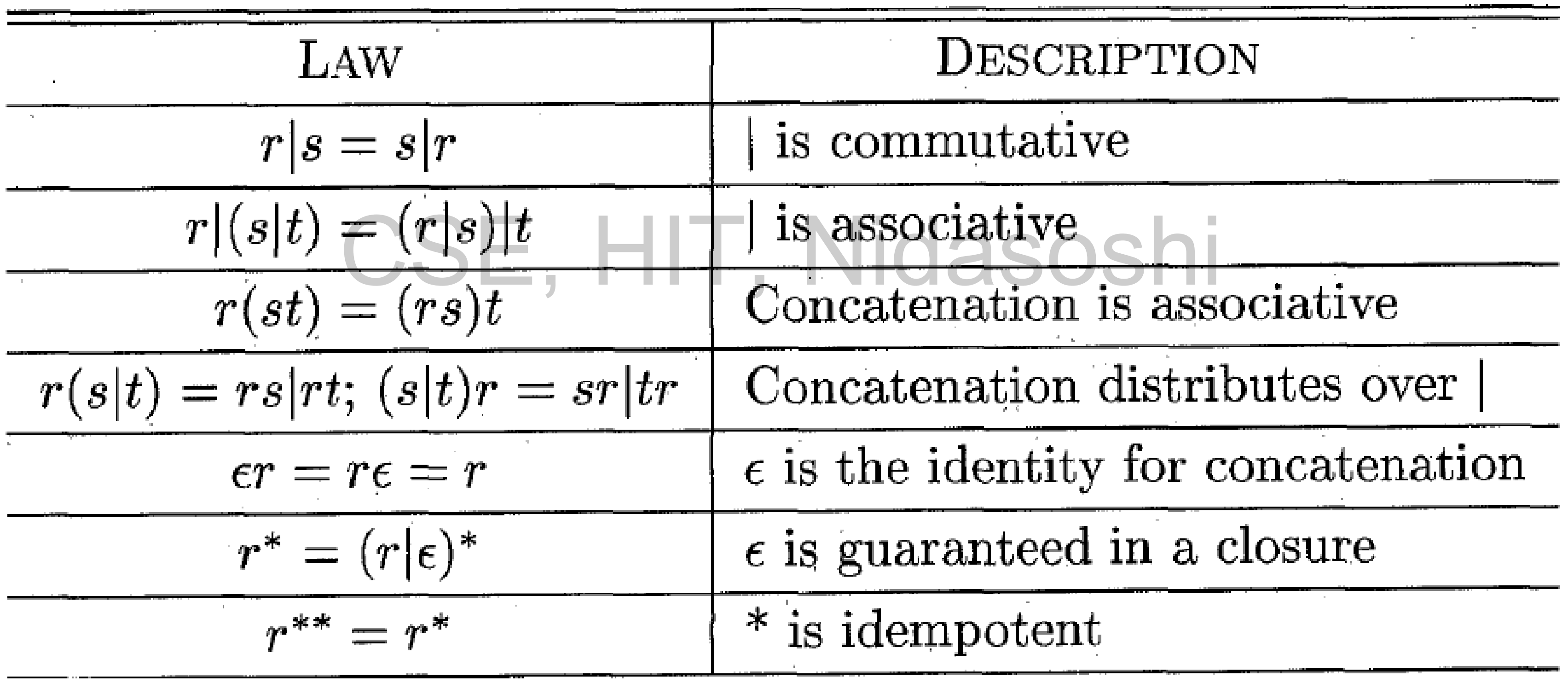
1. L and D are the alphabets of uppercase and lowercase letters and of digits.
2. L and D are languages, all of whose strings happen to be of length one.
3. L U D is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
4. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
5. L4 is the set of all 4-letter strings.
6. L* is the set of ail strings of letters, including e, the empty string.
7. L(L U D)* is the set of all strings of letters and digits beginning with a letter.
8. D+ is the set of all strings of one or more digits.

Specification of Tokens - Regular Expressions

- Any combination of languages with defined meaning is a regular expression.
- If r and s are two languages,
- Then,
- r|s is a regular expression which matches with either s or t.
- If letter contains alphabets (a-z or A-Z) and digit contains 0-9
- Then,
- letter_ (letter_ I digit)* is a regular expression which matches the identifier.

Specification of Tokens - Regular Expressions

- 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
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

Specification of Tokens - 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:

$$\begin{array}{ccc} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

- where:
- 1. Each di is a new symbol, not in C and not the same as any other of the d's, and
- 2. Each ri is a regular expression over the alphabet ∑ U {d1, d2,. . . , di-1).

Specification of Tokens - Regular Definitions

- **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.

$$\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{Z} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \mid \_ \\
\mathit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
\mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{array}$$

Specification of Tokens - Regular Definitions

- Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.
- The regular definition is a precise specification for this set of strings.

$$
\begin{array}{rcl}
digit & \rightarrow & 0 \mid 1 \mid \cdots \mid 9 \\
digits & \rightarrow & digit\ digit^{*} \\
optionalFraction & \rightarrow & .\ digits \mid \epsilon \\
optionalExponent & \rightarrow & (\ \mathtt{E}\ (\ + \mid - \mid \epsilon\ )\ digits\ ) \mid \epsilon \\
number & \rightarrow & digits\ optionalFraction\ optionalExponent
\end{array}
$$

Specification of Tokens - Regular Definitions

Extensions of Regular Expressions

1. One or more instances (+)
2. Zero or one instance (?)
3. Character classes [ ]

Specification of Tokens - Regular Definitions

Using these shorthands, we can rewrite the regular definition

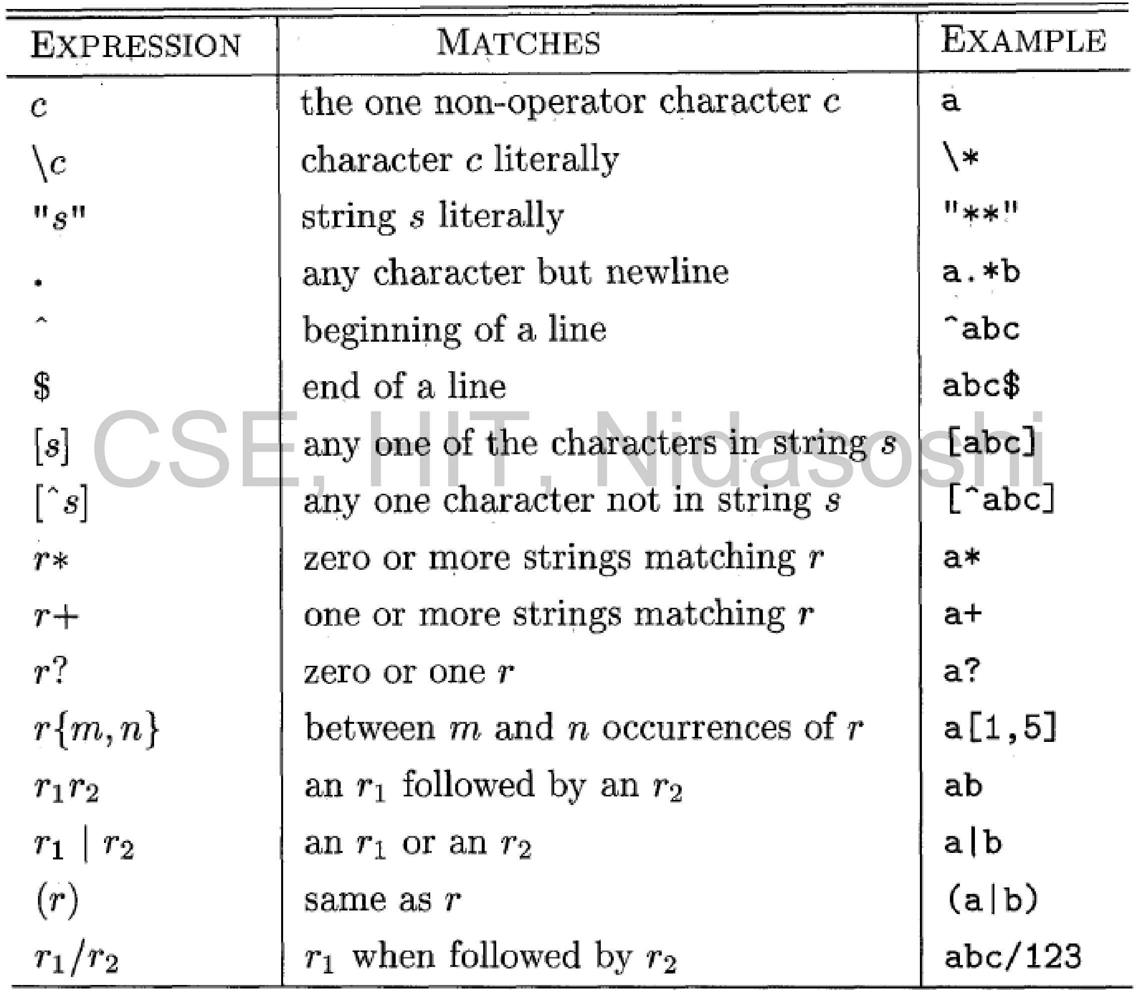
identifier as,

$$\begin{array}{rcl} letter\_ & \rightarrow & \texttt{[A-Za-z\_]} \\ digit & \rightarrow & \texttt{[0-9]} \\ id & \rightarrow & letter\_\ (\ letter\ |\ digit\ )^* \end{array}$$

Number as,

$$\begin{array}{rcl} digit & \rightarrow & \texttt{[0-9]} \\ digits & \rightarrow & digit^+ \\ number & \rightarrow & digits\ (\ .\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \end{array}$$

Specification of Tokens - Regular Definitions



EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
\\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`

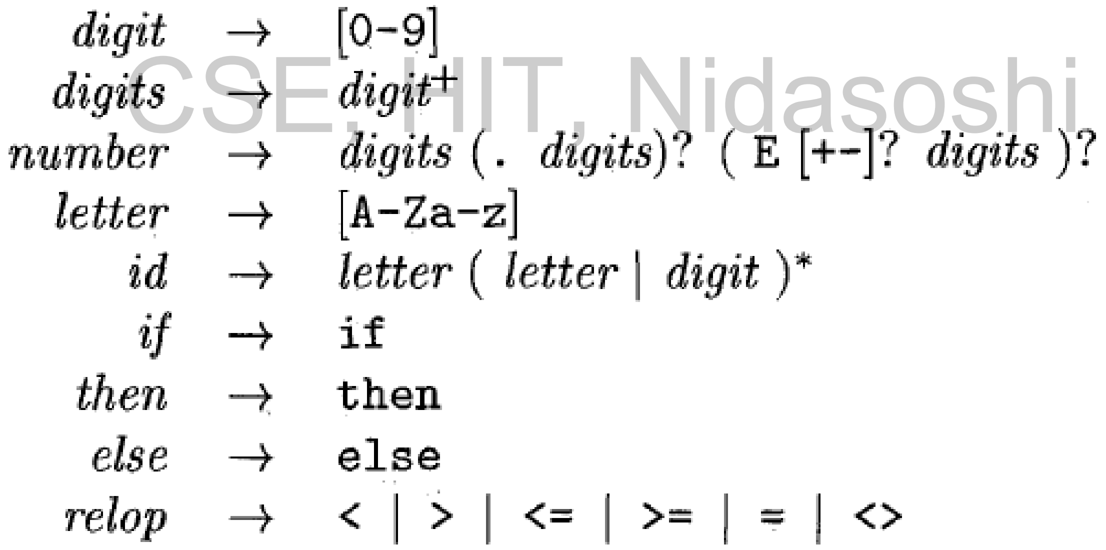
Recognition of Tokens

- Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.
- Example, A grammar for branching statements

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

Recognition of Tokens

- The terminals of the grammar, which are **if, then**, **else**, **relop**, **id**, and **number**, are the names of tokens as far as the lexical analyzer is concerned.
- The patterns for these tokens are described using regular definitions as shown below



digit	$\rightarrow$	`[0-9]`
digits	$\rightarrow$	$digit^+$
number	$\rightarrow$	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	$\rightarrow$	`[A-Za-z]`
id	$\rightarrow$	*letter* (*letter* | *digit*)*
if	$\rightarrow$	`if`
then	$\rightarrow$	`then`
else	$\rightarrow$	`else`
relop	$\rightarrow$	`<` | `>` | `<=` | `>=` | `=` | `<>`

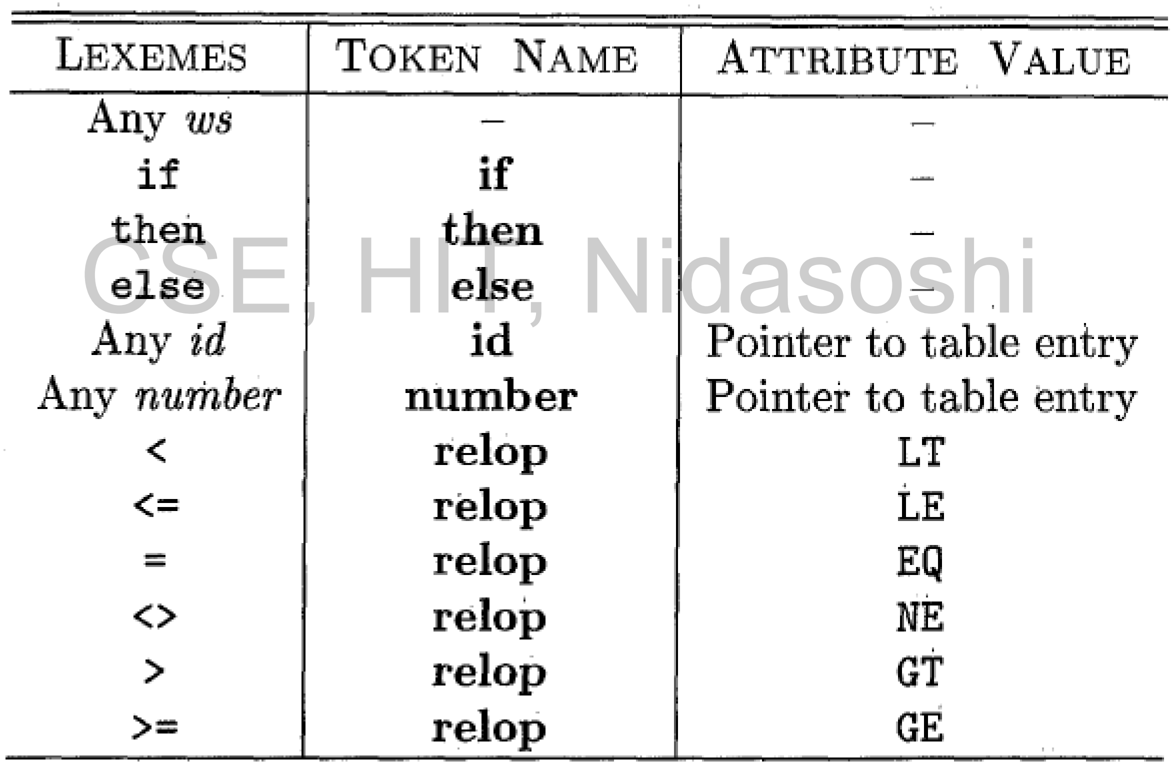
Recognition of Tokens

- In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

$$ws \rightarrow (\ \mathbf{blank} \mid \mathbf{tab} \mid \mathbf{newline}\ )^{+}$$

Recognition of Tokens

- Tokens, their patterns, and attribute values



LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Recognition of Tokens - Transition Diagrams

- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called **transition diagrams.**
- Transition diagrams have a collection of nodes or circles, called states.
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
- We may think of a state as summarizing all we need to know about what characters we have seen between the **lexemeBegin** pointer and the **forward** pointer

Recognition of Tokens - Transition Diagrams

- Edges are directed from one state of the transition diagram to another.
- Each edge is labeled by a symbol or set of symbols.
- If we are in some state s, and the next input symbol is a, we look for an edge out of state s labeled by a (and perhaps by other symbols, as well).
- If we find such an edge, we advance the forward pointer arid enter the state of the transition diagram to which that edge leads.
- We shall assume that all our transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels.

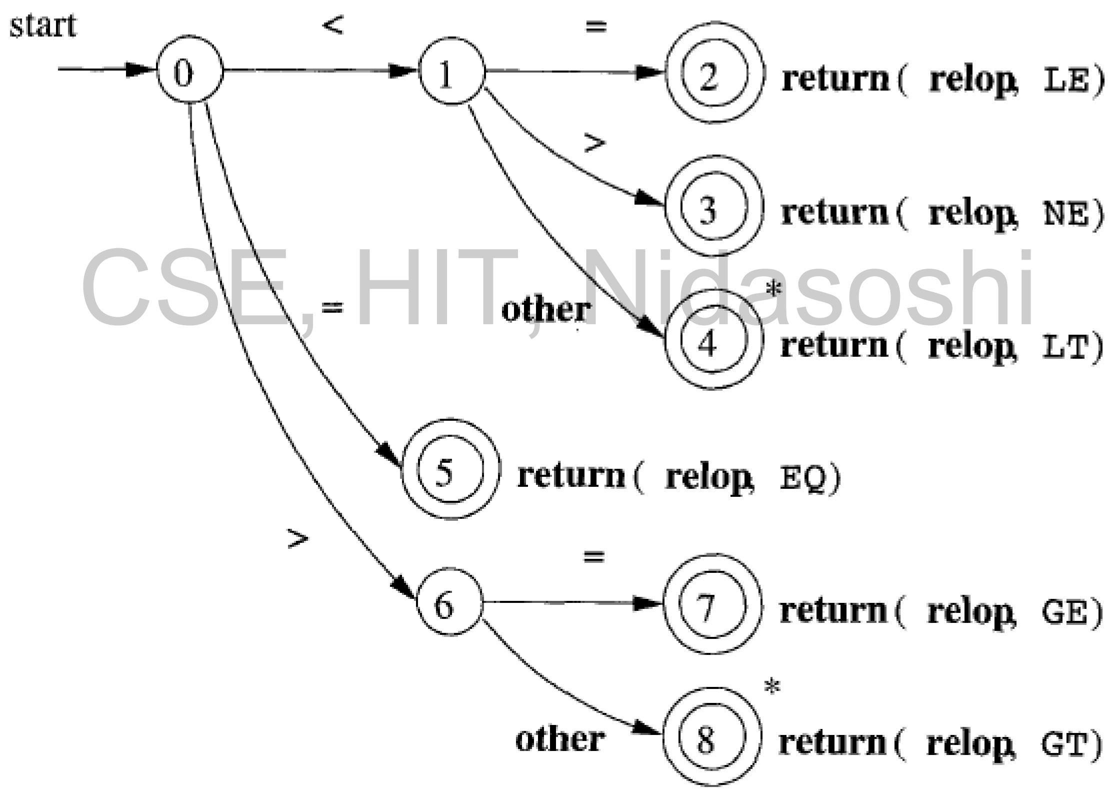
Recognition of Tokens - Transition Diagrams

Some important conventions about transition diagrams are:

1. Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the lexemeBegin and forward pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken - typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state.
2. In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting state.
3. One state is designated the start state, or initial state; it is indicated by an edge, labeled "start ," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Recognition of Tokens - Transition Diagrams

Transition diagram for **relop**



Recognition of Tokens - Transition Diagrams

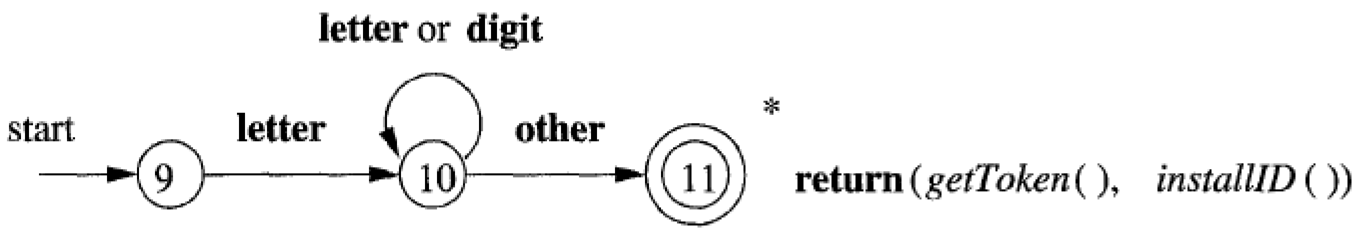
Recognition of Reserved Words and Identifiers,

- Recognizing keywords and identifiers presents a problem.
- Usually, keywords like if or then are reserved (as they are in our running example), so they are not identifiers even though they look like identifiers.

Recognition of Tokens - Transition Diagrams

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

1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents - either id or one of the keyword tokens that was initially installed in the table.

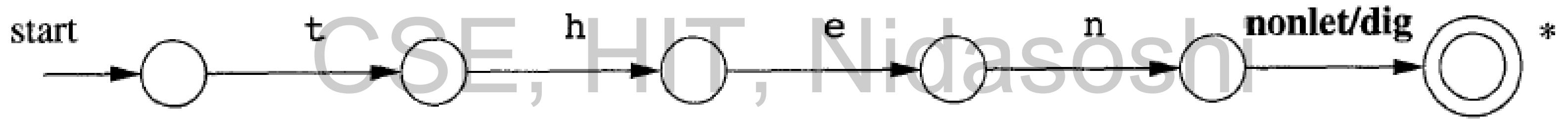


Recognition of Tokens - Transition Diagrams

2. Create separate transition diagrams for each keyword.Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a **"nonletter-or-digit,"** i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token **then** in situations where the correct token was id, with a lexeme like thenextvalue that has then as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.

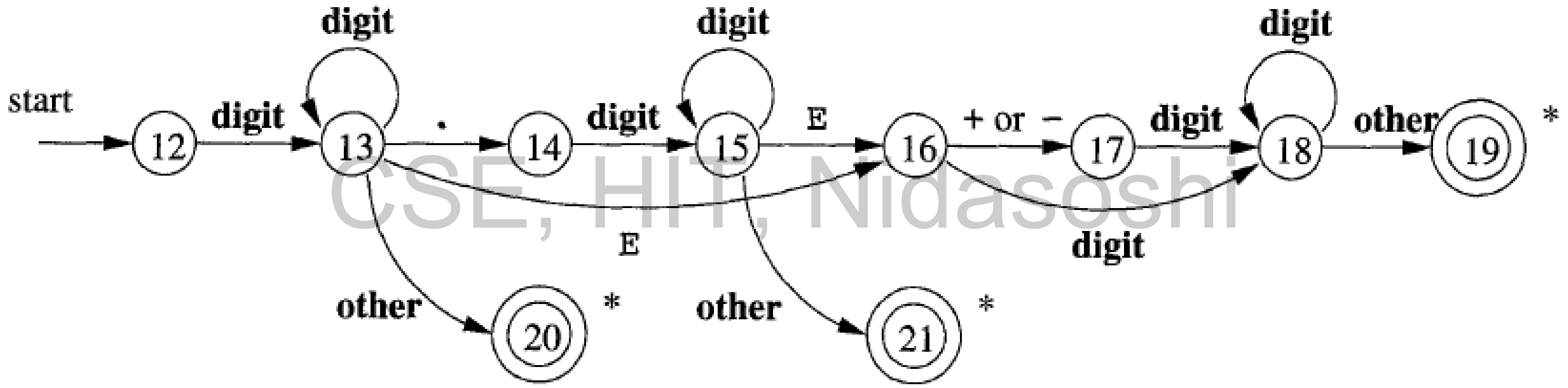
Recognition of Tokens - Transition Diagrams

Hypothetical transition diagram for the keyword **then**,



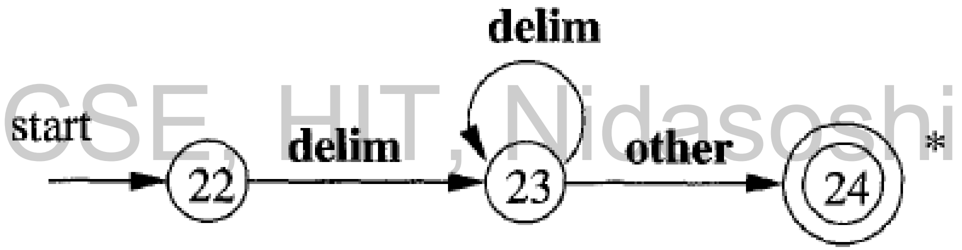
Recognition of Tokens - Transition Diagrams

The transition diagram for token **number,**



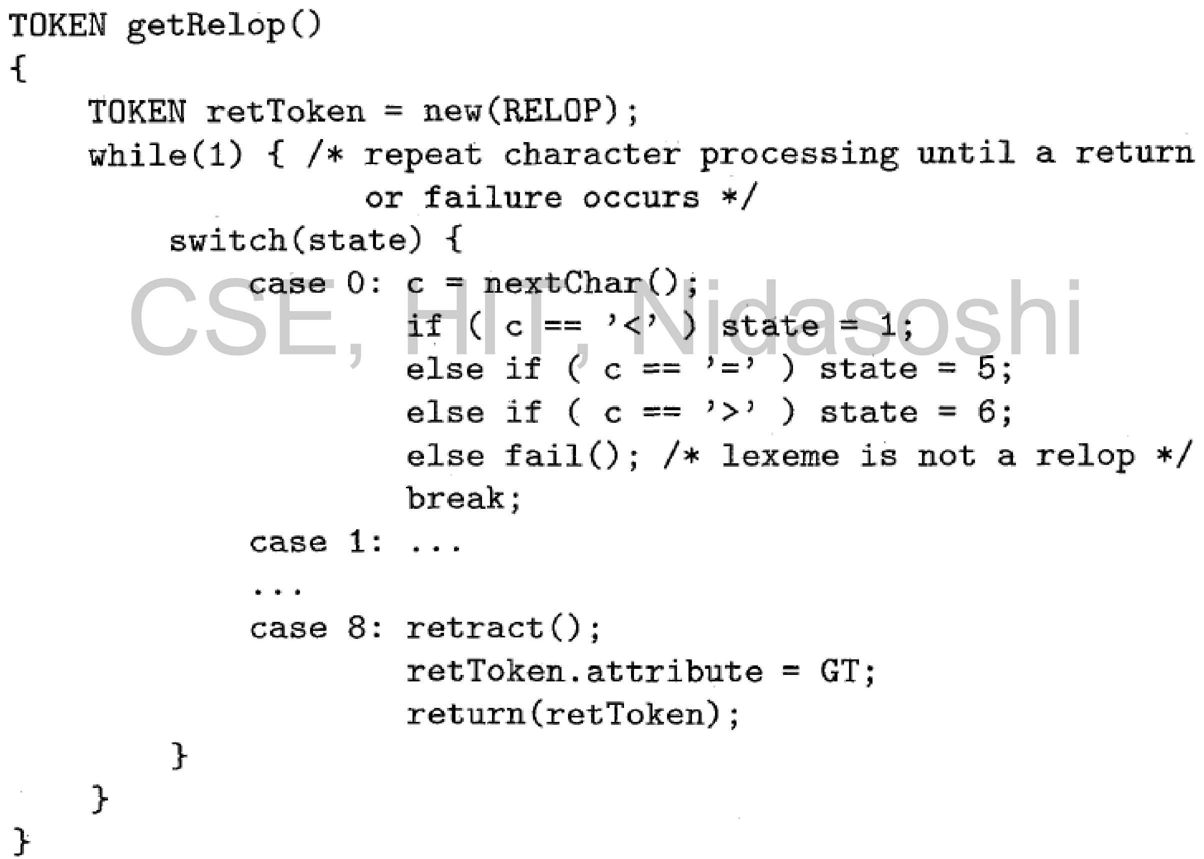
Recognition of Tokens - Transition Diagrams

A transition diagram for **whitespace**



Recognition of Tokens - Transition Diagrams

Sketch of implementation of **relop** transition diagram



```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Recognition of Tokens - Transition Diagrams

To place the simulation of one transition diagram in **three perspective**

1. We could arrange for the transition diagrams for each token to be tried sequentially.

Then, the function fail( ) resets the pointer forward and starts the next transition diagram, each time it is called.

This method allows us to use transition diagrams for the individual keywords.

We have only to use these before we use the diagram for **id,** in order for the keywords to be reserved words.

Recognition of Tokens - Transition Diagrams

2. We could run the various transition diagrams "in parallel," feeding the next input character to all of them and allowing each one to make whatever transitions it required.

If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input.

The normal strategy is to take the longest prefix of the input that matches any pattern.

That rule allows us to prefer identifier the next to keyword then, or the operator -> to -, for example.

Recognition of Tokens - Transition Diagrams

3. The preferred approach, and the one we shall take up in the following sections, is to combine all the transition diagrams into one. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern, as we discussed in item (2) above. In our running example, this combination is easy, because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for.

Thus, we could simply combine states 0, 9, 12, and **22** into one start state, leaving other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex, as we shall see shortly.