MODULE – 2

Language Processors and Lexical Analysis

Structure:

2.1 Introduction - Language Processors

A compiler is a program that can read a program in one language (source language) and translate it into an equivalent program in another language (target language). If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

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.

The machine-language target program produced by 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.

A language-processing system typically involves – preprocessor, compiler, assembler and linker/loader – in translating source program to target machine code.

2.2 Objectives

1. Study the structure of Compiler

2.3 The structure of a Compiler

Analysis: source program to intermediate representation (front end)
Synthesis: intermediate representation to target program (back end)

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 phases of a compiler are: lexical analyzer (scanning) (linear analysis), syntax analyzer (hierarchical analysis) (parsing), semantic analyzer, intermediate code generator, machine-independent code optimizer, code generator and machine-dependent code optimizer

Symbol table manager and error handler are two independent modules which will interact with all phases of compilation. A symbol table is a data structure containing a record for each identifier with fields for the attributes of the identifier. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. Each phase can encounter errors. After detecting an error, a phase must somehow deal with that error, so that compilation must proceed, allowing further errors in the source program to be detected.

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 de generation.
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.

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 ntax tree or the symbol table, for subsequent use during 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 variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

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. Different compiler construction tools are: parser generators, scanner generators, syntax-directed translation engines, code-generator generators, data-flow analysis engines, compiler-construction toolkits.

2.3.1 The Evolution of Programming Languages

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.

Impacts on compilers

Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users' applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

2.4 The Science of Building a Compiler

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled.
Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modeling in compiler design and implementation

The study of compilers is mainly study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

The science of code optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

2.5 Applications of Compiler Technology

Implementation of high-level programming languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

Optimizations for computer architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies.* Parallelism can be found at several levels: at the *instruction level,* where multiple operations are executed simultaneously and at the *processor level,* where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

*Parallelism-*All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

Memory Hierarchies- A memory hierarchy consists of several levels of storage with different speeds and sizes, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.

Design of new computer architectures

In the early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

Program translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages.

Software productivity tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs. An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Dataflow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

2.4.1 Programming Language Basics

The static/dynamic distinction among the most important issues that we face when designing a compiler for a language is what decisions the compiler can make about a program. If a compiler language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time.

Environments and states

Another important distinction we must make when discussing programming languages is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data. For example, the execution of an assignment such as **x = y +1** changes the value denoted by the name *x*. More specifically, the assignment changes the value in whatever location is denoted by *x*.

Static scope and block structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C + + , Java, and C # , also provide explicit control over scopes through the use of keywords like **public, private,** and **protected.** In this section we consider static-scope rules for a language with blocks, where a *block* is a grouping of declarations and statements. C uses braces {and} to delimit a block; the alternative use of **b e g in** and **end** for the same purpose dates back to *Algol.*

Explicit access control

Classes and structures introduce a new scope for their members. If *p* is an object of a class with a field (member) *x,* then the use of *x* in *p.x* refers to field *x* in the class definition. In analogy with block structure, the scope of a member declaration *x* in a class *C* extends to any subclass *C*, except if *C* has a local declaration of the same name *x.*

Dynamic scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope,* however, usually refers to the following policy: a use of a name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations.

Parameter passing mechanisms

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest.

Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

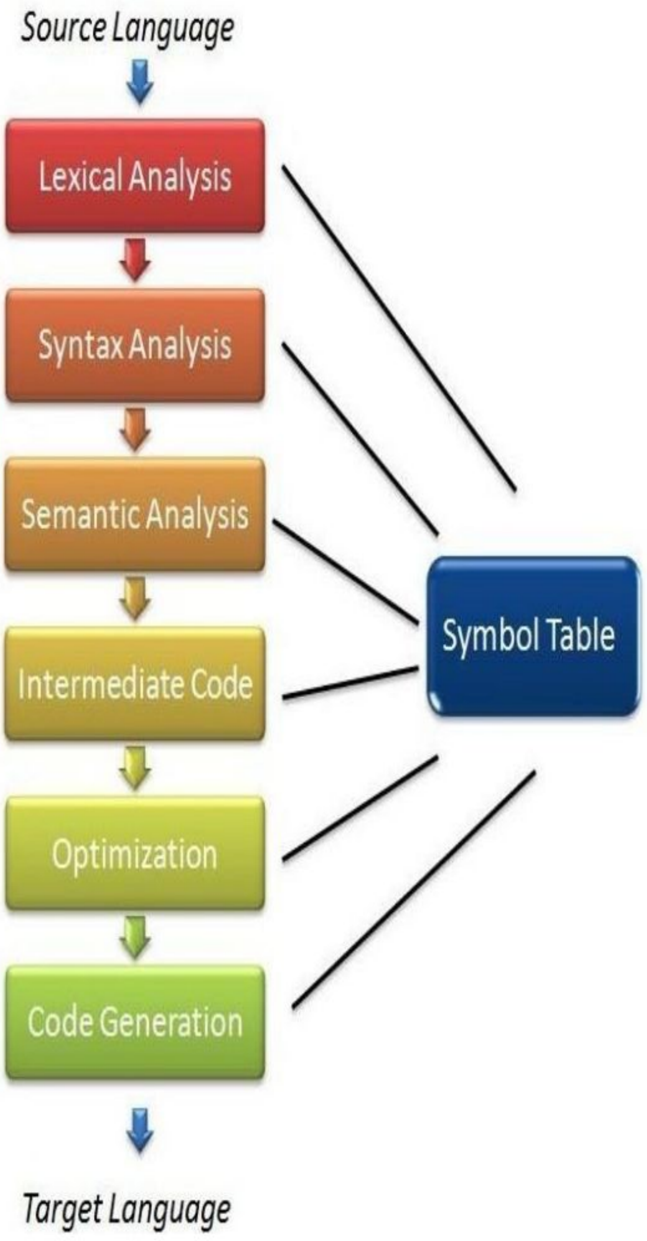
Lexical Analysis

What is Compiler?

- It is a program that translates one language to another.

Compiler phases

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate Code Generator
- Code Optimizer
- Code Generation



2.6 Lexical Analysis (scanner): The first phase of a compiler

Lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexeme
For each lexeme, the lexical analyzer produces a token of the form that it passes on to the subsequent phase, syntax analysis (token-name, attribute- value)
Token-name: an abstract symbol is used during syntax analysis. Attribute-value: points to an entry in the symbol table for this token. Tokens represent basic program entities such as: Identifiers, Literals, Reserved Words, Operators, Delimiters, etc.

Example Position = initial + rate * 60

1. "position" is a lexeme 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.

2. = is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. For notational convenience, the lexeme itself is used 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 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 (*).

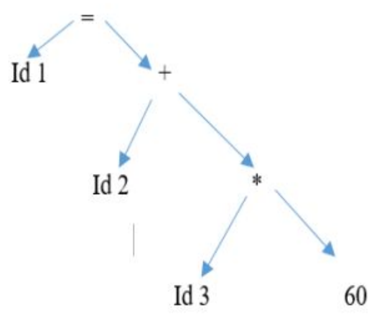
7. 60 is a lexeme that is mapped into the token (60)

Blanks separating the lexemes would be discarded by the lexical analyzer.

Syntax Analysis (parser) **:** The second phase of the compiler

- 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



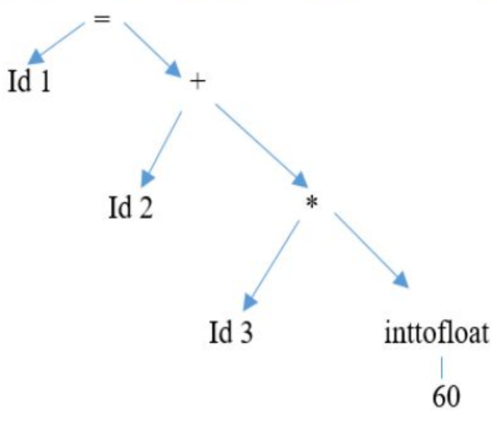
Semantic Analysis: Third phase of the compiler

The semantics of a program are its meaning as opposed to syntax or structure The semantics consist of:
Runtime semantics

- behavior of program at runtime

Static semantics–checked by the compile Static semantics include:

- Static semantics–checked by the compile
- Declarations of variables and constants before use
- Calling functions that exist (predefined in a library or defined by the user)
- Passing parameters properly
- Type checking.
- Annotates the syntax tree with type information



Intermediate Code Generation: three-address code

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.

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.

The considered intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

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

Code Optimization: to generate better target code
The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
Usually better means:
- Faster, shorter code, or target code that consumes less power.

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 int to float operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, t3 is used only once

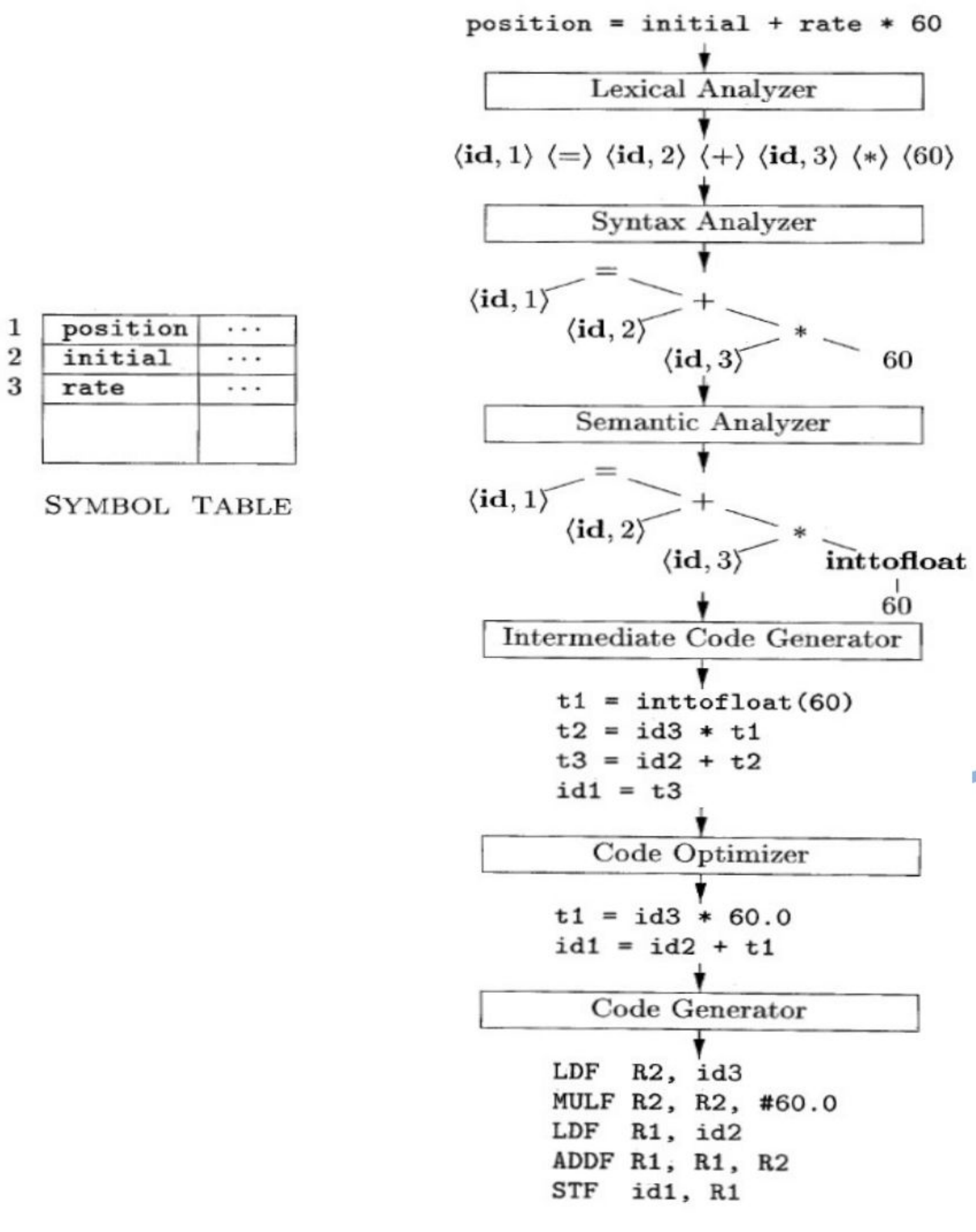
```
t1 = id3 * 60.0
id1 = id2 + t1
```

There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

Code Generation: 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
ADDF R1,  R1, R2
STF  id1, R1
```

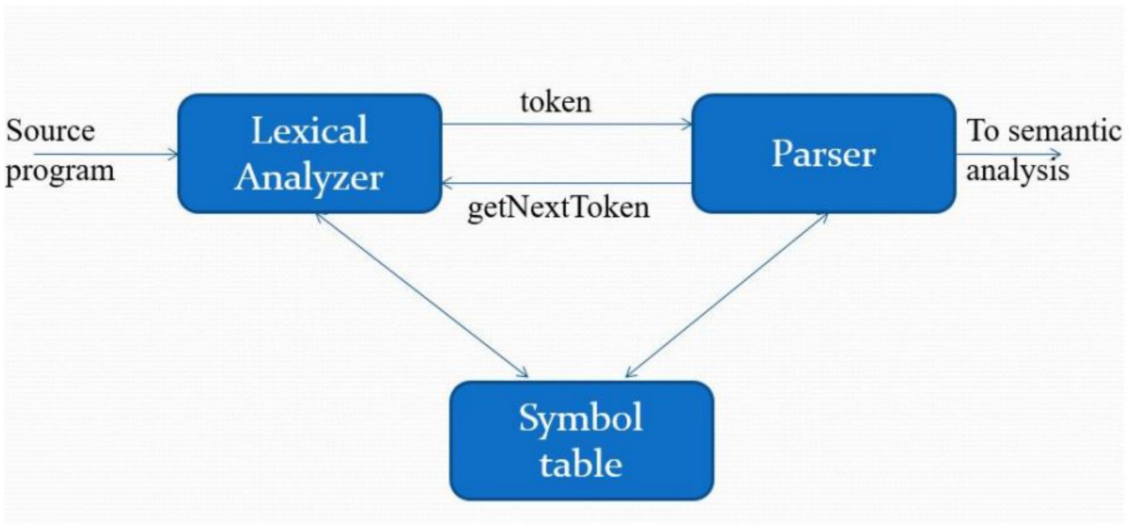


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
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
```

The role of lexical analyzer



Why to separate Lexical analysis and parsing

Simplicity of design

Improving compiler efficiency

Enhancing compiler portability

Tokens, Patterns and Lexemes

A token is a pair a token name and an optional token value

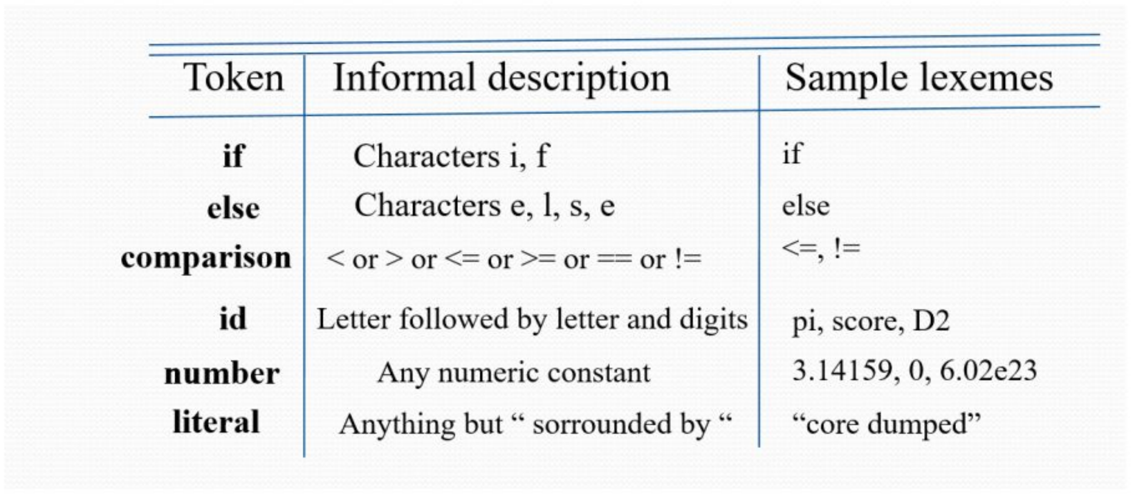
A pattern is a description of the form that the lexemes of a token may take

A lexeme is a sequence of characters in the source program that matches the pattern for a token

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token

Example



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 letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

Attributes for tokens

- **E = M * C ** 2**
    - **<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>**

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 f i is encountered for the first time in a C program in the context:

f i (a == f (x)) . ..

a lexical analyzer cannot tell whether f i is a misspelling of the keyword if or an undeclared function identifier. Since f i 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.

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.

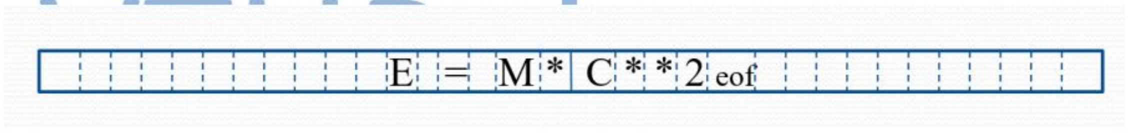
Other possible error-recovery actions are:
1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

Error recovery

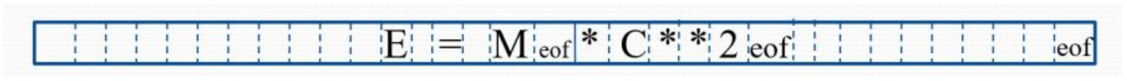
- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely



Sentinels



It is an extra key inserted at the end of the array. It is a special, dummy character that can't be part of source program. With respect to buffer pairs, the code for advancing forward pointer is:

```
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;
}
```

A *language* is any countable set of strings over some fixed alphabet. This definition is very broad. Abstract languages like 0, the *empty set,* or {e}, the set containing only the empty string, are languages under this definition. So too are the set of all syntactically well-formed C programs and the set of all grammatically correct English sentences, although the latter two languages are difficult to specify exactly.

Operations on languages

LUD: Union operation, where L=set of alphabets
{A..Z,a..z} and D=set of digits {0..9}
LD: Concatenation
L4:exponentiation: set of strings with 4 letters L0=
Li=Li-1L
L*=all strings ith : Kleene closure of L D+: set of
all strings of digits of one or more

- *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.
- *LD* is the set of strings of length two, each consisting of one letter followed by one digit.
- L4 is the set of all 4-letter strings.
- *L** is the set of all strings of letters, including e, the empty string.
- *L(L* U *D)** is the set of all strings of letters and digits beginning with a letter.
- *D+* is the set of all strings of one or more digits.

Specification of tokens

1. In theory of compilation regular expressions are used to formalize the specification of tokens
2. Regular expressions are means for specifying regular languages
3. Example:

   i. Letter_(letter_ | digit)*

4. Each regular expression is a pattern specifying the form of strings

Regular expressions

1. Ɛ is a regular expression, L(Ɛ) = {Ɛ}
2. If a is a symbol in ∑then a is a regular expression, L(a) = {a}
3. (r) | (s) is a regular expression denoting the language L(r) L(s)
4. (r)(s) is a regular expression denoting the language L(r)L(s)

5. (r)* is a regular expression denoting (L(r))*
6. (r) is a regular expression denoting L(r)

➢ Regular definitions

1. d1 -> r1
2. d2 -> r2
3. …
4. dn -> rn
5. Example:
6. letter_ -> A | B | … | Z | a | b | … | Z | _
7. digit ->0|1|…|9
8. id -> letter_ (letter_ | digit)*

Extensions

- One or more instances: (r)+
- Zero of one instances: r?
- Character classes: [abc]
- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter_|digit)*

Transition diagrams

These are the flow charts, as an intermediate step in the construction of a lexical analyzer. This takes actions when a lexical analyzer is called by the parser to get the next token. We use transition diagram to keep track of information about characters that are seen as and when the forward pointer scans the input. Lexeme beginning pointer points to the character following the last lexeme found.

E=M* C**2eof

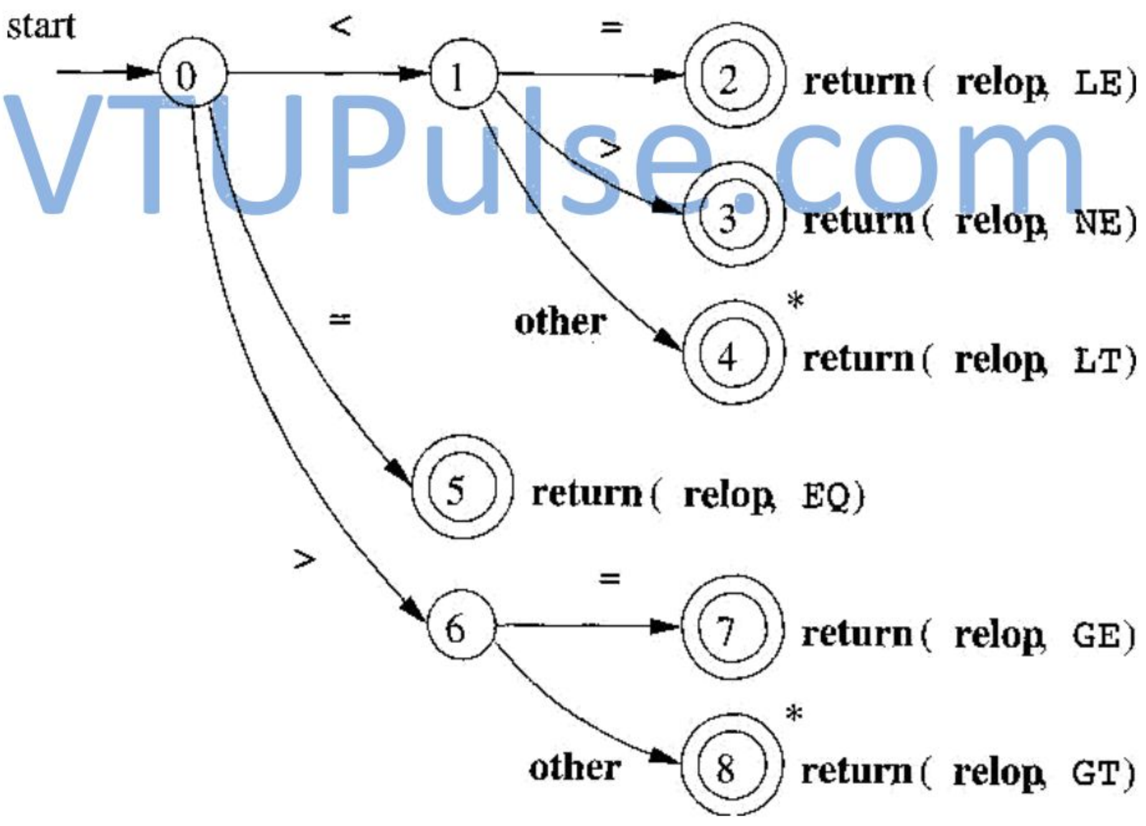
Transition diagrams have a collection of nodes or circles, called *states.* Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of at ate as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer.

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**,** 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 as assume that 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. We shall relax the condition of determinism, making life much easier for the designer of a lexical analyzer, although trickier for the implementer. 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.


start
<
=
>
other
=
>
=
other
*
*
return(relop, LE)
return(relop, NE)
return(relop, LT)
return(relop, EQ)
return(relop, GE)
return(relop, GT)