UNIT –III

SOURCE PROGRAM ANALYSIS

P a g e | **1** LANGUAGE TRANSLATORS

**2 MARKS**

**1. What is a compiler? (MAY, NOV 2012)**

A *compiler* is a program that reads a program written in one language – the source language and translates it into an equivalent program in another language – the target language. In translation process, the compiler reports to its user the presence of errors in the source program.

Source program Target program



Compiler

Error messages

1. **What are the classifications of a compiler?**

Compilers are sometimes classified as:

* + Single-pass
	+ Multi-pass
	+ Load-and-go
	+ Debugging or
	+ Optimizing
1. **What are the two parts of a compilation? Explain briefly.**

There are two parts to compilation as

* 1. Analysis part
	2. Synthesis part
* The *analysis part* breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
* The *synthesis part* constructs the desired target program from the intermediate representation.

**4. What are the tools available in analysis phase?**

Many software tools that manipulate source programs are

* + Structure editors
	+ Pretty printers
	+ Static checkers
	+ Interpreters
1. **What is Query Interpreters?**

A *Query interpreter* translates a predicate containing relational and Boolean operators into commands to search a database for records satisfying that predicate.

P a g e | **2** LANGUAGE TRANSLATORS

1. **List the analysis of the source program?**

Analysis consists of three phases:

* + - Linear Analysis.
		- Hierarchical Analysis.
		- Semantic Analysis.
1. **What is linear analysis?**
	* In a compiler, *linear analysis* is called *lexical analysis* or *scanning.*
	* *Linear analysis* in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.
2. **What is hierarchical analysis? (NOV 2011)**
	* *Hierarchical analysis* is called *parsing* or *syntax analysis*.
	* *Hierarchical analysis* involves grouping the tokens of the source program into grammatical phases thatare used by the compiler to synthesize output.
3. **What is semantic analysis?**
	* The *Semantic analysis*, it checks the source program for semantic errors and gathers type information for the subsequent code generation phase.
	* It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements.
4. **Draw the parse tree for a source program as position: = initial + rate \* 60.**



P a g e | **3** LANGUAGE TRANSLATORS

1. **List the various phases of a compiler.**

The following are the various phases of a compiler:

* + - Lexical Analyzer
		- Syntax Analyzer
		- Semantic Analyzer
		- Intermediate code generator
		- Code optimizer
		- Code generator
1. **What is a symbol table?**
	* A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store

or retrieve data from that record quickly.

* + Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.
1. **What is Intermediate code generator?**

The intermediate representation should have two important properties;

* + - it should be easy to produce, and
		- it should be easy to translate into the target program.
1. **What is three address code?**
	* An intermediate form called “*three-address code*, “which is like the assembly language for a machine in which every memory location can act like a register.
	* Three-address code consists of a sequence of instructions, each of which has at most three operands.
2. **What is code generator?**
	* The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.
	* Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.
3. **Mention the cousins of the compiler?**

Cousins of the compiler are:

* Preprocessor
* Compiler
* Assembler
* Two-Pass Assembly
* Loader and Link-Editor

P a g e | **4** LANGUAGE TRANSLATORS

**17. State with example the cousins of compilers. (NOV 2013)**

Skeletal source program

↓

**Preprocessor**

↓

Source program

↓

**Compiler**

↓

Target assembly program

↓

**Assembler**

↓

Relocatable machine code

↓

|  |  |  |
| --- | --- | --- |
|  | Library, relocatable object files |  |
| **Loader/ link editor** |  |
|  |  |
|  |  |  |



↓

Absolute machine code

**18. What is Preprocessors? List its functions.**

Preprocessors produce input to compilers. They may perform the following functions:

* + - Macro Processing
		- File inclusion
		- Rational Preprocessors
		- Language extensions
1. **Define Assembly code?**
	* Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses.
	* A typical sequence ***b := a + 2****,*the assembly instructions might be

***MOV a, R1***

***ADD #2, R1***

***MOV R1, b***

**20. What is the use Two-Pass assembly?**

The assembler makes two passes over the input, where a *pass* consists of reading an input file once.

* In the *first pass*, all the identifiers that denote storage locations are found and stored in a symbol table.
* Identifiers are assigned storage locations as they are encountered for the first time.
* In the *second pass*, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifier representing a location into address given for that identifier in the symbol table.
* The output of the second pass is usually relocatable machine code.

P a g e | **5** LANGUAGE TRANSLATORS

1. **What is loader and link-editor?**
	* Usually, a program called a loader performs the two functions of *loading* and *link-editing.*
	* The process of *loading* consists of taking relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper location.
	* The *link-editor* allows us to make a single program from several files of relocatable machine code. These files may be the result of several different compilation and one or more library files of routines

provided by the system.

1. **State the function of front end and back end of a compiler phase. (MAY 2013)**

The front end consists of those phases that depends primarily on the source language and are largely independent of the target machine.

These include

* + Lexical analysis
	+ Syntactic analysis
	+ Semantic analysis
	+ Creation of symbol table
	+ Generation of intermediate code
	+ Code optimization
	+ Error handling
1. **State the function back end of a compiler phase. (MAY 2013)**

The back end of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language.

These include

* + Code optimization
	+ Code generation
	+ Error handling and
	+ Symbol-table operations
1. **What is single pass?**

Several phase of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file.

**25. Define compiler-compiler.**

Systems to help with the compiler-writing process are often been referred to as *compiler-compilers,* *compiler-generators or translator-writing systems.* Largely they are oriented around a particular model oflanguages, and they are suitable for generating compilers of languages similar model.

P a g e | **6** LANGUAGE TRANSLATORS

1. **List the various compiler construction tools.**

The various compiler construction tools are

* + - Parser generators
		- Scanner generators
		- Syntax-directed translation engines
		- Automatic code generators
		- Data-flow engines
1. **What is the role of lexical analyzer? (NOV 2013)**
	* The *lexical analyzer* is the first phase of a compiler.
	* Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses the syntax analysis.
	* Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.



1. **What are the issues in lexical analysis?**

The issues in lexical analysis are

* + - * Simpler design is the most important consideration.
			* Compiler efficiency is improved.
			* Compiler portability is enhanced.
1. **Why separate lexical analysis phase is required? (MAY 2013)**
	1. Simpler design is the most important consideration.
		* Comments and white space have already been removed by lexical analyzer.
	2. Compiler Efficiency is improved.
		* Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.
	3. Compiler Portability is enhanced.
		* Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

P a g e | **7** LANGUAGE TRANSLATORS

1. **What is the major advantage of a lexical analyzer generator? (NOV 2011)**

The major advantages of a lexical analyzer generator are

* + - One task is stripping out from the source program comments and white space in the form of blank, tab and new line characters.
		- Another is error messages from the compiler in the source program.
1. **What are tokens, patterns, and lexeme?**
	* *Tokens*- Sequence of characters that have a collective meaning.
	* *Patterns*- There is a set of strings in the input for which the same token is produced as output. This setof strings is described by a rule called a pattern associated with the token.
	* *Lexeme*- A sequence of characters in the source program that is matched by the pattern for a token.
2. **Differentiate between tokens, patterns, and lexeme?**

|  |  |  |
| --- | --- | --- |
| **Token** | **lexeme** | **patterns** |
|  |  |  |
| const | const | const |
|  |  |  |
| if | if | if |
|  |  |  |
| relation | <, <=, =, < >, >, >= | < or <= or = or < > or >= or > |
|  |  |  |
| id | pi, count, D2 | letter followed by letters and digits |
|  |  |  |
| num | 3.1416, 0, 6.02E23 | any numeric constant |
|  |  |  |
| literal | “core dumped” | any characters between “ and “ except “ |
|  |  |  |

1. **List the various Panic mode recovery in lexical analyzer?**

Possible error recovery actions are:

* 1. Deleting an extraneous character
	2. Inserting a missing character
	3. Replacing an incorrect character by a correct character
	4. Transposing two adjacent characters
1. **What are the approaches to implement lexical analyzer?**

The three general approaches to the implementation of a lexical analyzer

1. Use a lexical analyzer generator such as the Lex compiler to produce a regular expression based specification. In this case, the generator provides for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems programming languages using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and reading of input.

P a g e | **8** LANGUAGE TRANSLATORS

**35. What is input buffering?**

The *input buffer* is useful when look-ahead on the input is to identify tokens.

1. **Define sentinels.**
	* *Sentinel* is a special character that cannot be part of the source program.
	* The techniques for speeding up the lexical analyzer, use the “sentinels “ to mark the buffer end.
2. **List the operations on languages.**

The operations that can be applied to languages are

* + - Union - L U M ={s | s is in L or s is in M}
		- Concatenation – LM ={st | s is in L and t is in M}
		- Kleene Closure – L\* (zero or more concatenations of L)
		- Positive Closure – L+ (one or more concatenations of L)
1. **Write a regular expression for an identifier.**
	* An identifier is defined as a letter followed by zero or more letters or digits. The regular expression for an identifier is given as

**letter (letter | digit)\***

1. **How the regular expression can be defined in specification of the language. 1.** is a regular expression that denotes {}, the set containing empty string.

**2.** If*a*is a symbol in, then*a*is a regular expression that denotes {*a*}, the set containing the string*a*. **3.** Suppose*r*and*s*are regular expressions denoting the language L(*r*) and L(*s*), then

* + - (*r*) |(*s*) is a regular expression denoting L(*r*) L(*s*).
		- (*r*)(*s*) is regular expression denoting L (*r*) L(*s*).
		- (*r*) \* is a regular expression denoting (L (*r*))\*.
		- (*r*) is a regular expression denoting L (*r*).
1. **Mention the various notational short hands for representing regular expressions.**
	* + One or more instances
		+ Zero or one instance
		+ Character classes
		+ Non regular sets
2. **What is transition diagram? (NOV 2012)**
	* An intermediate step in the construction of a lexical analyzer, we first produce a stylized flowchart called a *transition diagram.*
	* Transition diagram depicts the actions that take place when a lexical analyzer is called by the parser to get the next token.

P a g e | **9** LANGUAGE TRANSLATORS

**42. Define Lex complier?**

A particular tool called *Lex*, used to specify lexical analyzers for a variety of languages. We refer to the tool as the *Lex compiler* and to its input specification as the Lex language.

**43. How to create a lexical analyzer with Lex?**



**44. List out the parts on Lex specifications. (MAY 2012)**

A Lex program consists of three parts: declarations

%%

translation rules

%%

auxiliary procedures

* The declarations section includes declarations of variables, manifest constants, and regular definitions.
* The translation rules of a Lex program are statement of the form

*p*1{ *action1*} *p*2{ *action2*}

…… …..

*p*n{ *actionn*}

* The auxiliary procedures are needed by the actions. These procedures can be compiled separately and loaded with the lexical analyzer.

P a g e | **10** LANGUAGE TRANSLATORS

* 1. **MARKS**
1. **Write a short note on compiler design? (6 marks)**

A ***compiler*** is a program that reads a program written in one language – the source language and translates it into an equivalent program in another language – the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

Source program Target program



***Compiler***

Error messages

* At first, the variety of compilers may appear overwhelming. There are thousands of source languages, ranging from traditional programming languages such as FORTRAN and Pascal to specialized languages in

every area of computer application.

* Target languages are equally as varied; a target language may be another programming language, or the machine language of any computer between a microprocessor and a supercomputer.

Compilers are sometimes classified as:

* + Single-pass
	+ Multi-pass
	+ Load-and-go
	+ Debugging or
	+ Optimizing
* Throughout the 1950’s, compilers were considered notoriously difficult programs to write. The first

FORTRAN compiler, for example, took 18 staff-years to implement.

**Analysis-Synthesis Model of Compilation:**

There are two parts to compilation as

* 1. Analysis part
	2. Synthesis part
* The *analysis part* breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
* The *synthesis part* constructs the desired target program from the intermediate representation.

P a g e | **11** LANGUAGE TRANSLATORS

* During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a ***tree.***
* Often, a special kind of tree called a ***syntax tree*** is used, in syntax tree each node represents an operation and the children of the node represent the arguments of the operation.

For example, a syntax tree of an assignment statement is **position: = initial + rate \* 60.**



Many software tools that manipulate source programs first perform some kind of analysis. Some examples of such tools include:

* Structure editors
* Pretty printers
* Static checkers
* Interpreters

**Structure editors:**

A *structure editor* takes as input a sequence of commands to build a source program. The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, appropriate hierarchical structure on the source program. It is useful in the preparation of source program.

**Pretty printers:**

A *pretty printer* analyzes a program and prints it in a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font.

**Static checkers:**

*A static checker* reads a program, analyzes it, and attempts to discover potential bugs withoutrunning the source program.

**Interpreters:**

*Interpreters* are frequently used to execute command languages, since each operator executed incommand languages is usually a complex routine such as an editor or compiler.

P a g e | **12** LANGUAGE TRANSLATORS

The analysis portion in each of the following examples is similar to that of a conventional compiler.

* Text formatters
* Silicon compilers
* Query interpreters

**Text formatters:**

A *text formatter* takes input that is a stream of characters, most of which is text to be typeset, and it includes commands to indicate paragraphs, figures, or mathematical structures like subscripts and the superscripts.

**Silicon compilers:**

A *silicon compiler* has a source language that is similar or identical to a conventional programming language. However, the variables of the language represent, not locations in memory, but also logical signals (0 or 1) or groups of signals in a switching circuit. The output is a circuit design in an appropriate language.

**Query interpreters:**

A *Query interpreter* translates a predicate containing relational and Boolean operators into commands to search a database for records satisfying that predicate.

1. **Explain the analysis of the source program? (11 marks)**

In compiling, analysis consists of three phases:

* + Linear Analysis
	+ Hierarchical Analysis
	+ Semantic Analysis

**Linear Analysis:**

* *Linear analysis*, in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning
* In a compiler, linear analysis is called lexical ***analysis or scanning.***

For example, in lexical analysis the characters in the assignment statement **position: = initial + rate \* 60** would be grouped into the following tokens:

1. The identifier position.
2. The assignment symbol :=
3. The identifier initial.
4. The plus sign.
5. The identifier rate.
6. The multiplication sign.
7. The number 60.

P a g e | **13** LANGUAGE TRANSLATORS

* The blanks separating the characters of these tokens would normally be eliminated during the lexical analysis.

**Hierarchical Analysis:**

* *Hierarchical analysis* is called***parsing*****or*****syntax analysis*.**
* Hierarchical analysis involves grouping the tokens of the source program into grammatical phases that are used by the compiler to synthesize output.
* The grammatical phrases of the source program are represented by a parse tree.



**Parse tree for position: = initial + rate \* 60**

* + In the expression **initial + rate \* 60**, the phrase rate \* 60 is a logical unit because the usual conventions of arithmetic expressions tell us that the multiplication is performed before addition.
	+ Because the expression **initial + rate** is followed by a \*, it is not grouped into a single phrase by itself.
	+ The hierarchical structure of a program is usually expressed by *recursive rules*. For example, the following rules, as part of the definition of expression:
		1. Any *identifier* is an expression.
		2. Any *number* is an expression
		3. If *expression1* and *expression2* are expressions, then so are
			- *expression1* + *expression2*
			- *expression1* \* *expression2*
			- (*expression1*)
* Rules (1) and (2) are non-recursive basic rules, while (3) defines expressions in terms of operators applied to other expressions.

P a g e | **14** LANGUAGE TRANSLATORS

Similarly, many languages define statements recursively by rules such as: **1.** If*identifier1*is an identifier and*expression2*is an expression, then

*identifier1 := expression2* is a statement.

**2.** If*expression1*is an expression and*statement2*is a statement, then ***while*** *(expression1)* ***do*** *statement2*

***if*** *(expression1)* ***then*** *statement2*are statements.

A ***syntax tree*** is a compressed representation of the parse tree in which the operators appear as the interior nodes and the operands of an operator are the children of the node for that operator.



**Semantic Analysis:**

* The *semantic analysis* phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
* It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operand of expressions and statements.
* An important component of semantic analysis is *type checking.*
* The compiler checks that each operator has operands that are permitted by the source language specification.
* For example, when a binary arithmetic operator is applied to an integer and real.
* In this case, the compiler may need to be converting the integer to a real. As shown in figure given below.



P a g e | **15** LANGUAGE TRANSLATORS

**3. Explain the various phases of a compiler with an example? (11 marks) (NOV 2011, 2013) (MAY, NOV 2012)(MAY 2013)**

A compiler operates in phases, each of which transforms the source program from one representation to another. The six phases of a complier are

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate code generator
5. Code optimizer
6. Code generator

Two other activities are

* Symbol table Manager
* Error handler

A typical decomposition of a compiler is shown in fig given below



**Phases of a compiler**

P a g e | **16** LANGUAGE TRANSLATORS

**The Analysis Phases:**

As translation progresses, the compiler’s internal representation of the source program changes. Consider the translation of the statement,

**position: = initial + rate \* 10**

**Lexical analyzer:**

* The ***lexical analysis*** phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an

identifier, a keyword (if, while, etc.), a punctuation character or a multi-character operator like :=.

* In a compiler, *linear analysis* is called ***lexical analysis*** **or** ***scanning.***
* ***Linear analysis*** in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters.
* The character sequence forming a token is called the ***lexeme*** for the token.
* Certain tokens will be augmented by a ‘*lexical value’*. For example, when an identifier like *rate* is found, the lexical analyzer not only generates a token id, but also enters the lexeme rate into the symbol table,

if it is not already there.

Consider **id1, id2** and **id3** for **position, initial, and rate** respectively, that the internal representation of an identifier is different from the character sequence forming the identifier.

The representation of the statement given above after the lexical analysis would be: **id1: = id2 + id3 \* 10**

**Syntax analyzer:**

* ***Hierarchical analysis*** involves grouping the tokens of the source program into grammatical phasesthat are used by the compiler to synthesize output.
* *Hierarchical analysis* is called***parsing*****or*****syntax analysis.***
* Syntax analysis imposes a hierarchical structure on the token stream, which is shown by syntax trees.



P a g e | **17** LANGUAGE TRANSLATORS

**Semantic analyzer:**

* The ***Semantic analysis,*** it checks the source program for semantic errors and gathers type information for the subsequent code generation phase.
* It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements.
* Compiler report an error, if **real number** is used to **index** an array.
* The bit pattern representing an integer is generally different from the bit pattern for a real, even they have the same value.
* For example, the identifiers position, initial, rate declared to be **real** and that 60 by itself assumed to be **integer.**
* The general approach is to convert the integer to a real.



**Intermediate code generator:**

* After Syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. The intermediate representation 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 program.
* An intermediate form called “*three-address code*, “which is like the assembly language for a machine in which every memory location can act like a register.
* Three-address code consists of a sequence of instructions, each of which has at most three operands.
* The source program might appear in three-address code as

**temp1: = inttoreal (60) temp2: = id3 \* temp1 temp3: = id2 + temp2 id1: = temp3**

P a g e | **18** LANGUAGE TRANSLATORS

This intermediate form has several properties:

1. First, each three address instruction has at most one operator in addition to the assignment. Thus, when generating these instructions, the compiler has to decide on the order in which operations are to be done; the multiplication precedes the addition in the source program.
2. Second, the compiler must generate a temporary name to hold the value computed by each instruction.
3. Third, some “three-address” instructions have fewer than three operands.

**Code Optimization:**

* The *code optimization phase* attempts to improve the intermediate code, so that faster-running machine code will result.
* For example, a natural algorithm generates the intermediate code, using an instruction for each operator in the tree representation after semantic analysis, even though there is a better way to perform the same calculation, using the two instructions.

**temp1 := id3 \* 60.0 id1 := id2 + temp1**

* There is nothing wrong with this simple algorithm, since the problem can be fixed during the code-optimization phase.
* That is, the compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the inttoreal operation can be eliminated.
* There is a great variation in the amount of code optimization different compilers.
* ‘Optimizing compilers’, a significant fraction of the time of the compiler is spent on this phase.

**Code Generation:**

* The final phase of the compiler is the *generation of target code*, consisting normally of relocatable machine code or assembly code.
* Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.
* The assignment of variables to registers.

For example, using registers 1 and 2, the translation of the code as

**MOVF id3, R2**

**MULF #60.0, R2**

**MOVF id2, R1**

**ADDF R2, R1**

**MOVF R1, id1**

P a g e | **19** LANGUAGE TRANSLATORS

* The first and second operands of each instruction specify a source and destination, respectively.
* The ***F*** in each instruction tells us that instructions deal with floating-point numbers.
* This code moves the contents of the address id3 into register 2, and then multiplies it with the real-constant 60.0.
* The ***#*** signifies that 60.0 is to be treated as a constant.
* The third instruction moves id2 into register 1 and adds to it the value previously computed in register 2.
* Finally, the value in register 1 is moved into the address of id1.

**Symbol Table Management**:

* An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier.
* These attributes may provide information about the storage allocated for an identifier, its type, its scope and in case of procedure names, such things at the number and types of its arguments and methods of passing each argument and type returned.
* The ***symbol table*** is a data structure containing a record for each identifier with fields for the attributes of the identifier.
* The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.
* Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.
* However, the attributes of an identifier cannot normally be determined during lexical analysis. For example, in a ***Pascal declaration*** like

***var position, initial, rate : real;***

* The type real is not known when position, initial and rate are seen by the lexical analyzer.
* The remaining phases get information about identifiers into the symbol table and then use this information in various ways.

**Error Detection and Reporting:**

* Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
* A compiler that stops when it finds the first error is not as helpful as it could be.
* The *syntax and semantic analysis* phases usually handle a large fraction of the errors detectable by the compiler.

P a g e | **20** LANGUAGE TRANSLATORS

* The *lexical phase* can detect errors where the characters remaining in the input do not form any token of the language.
* Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.
* During semantic analysis the compiler tries to detect the right syntactic structure but no meaning to the operation involved.
* e.g. we try to add two identifiers, one of which is the name of an array and the other the name of the procedure.

**Example:**

Consider the translation of the statement, **position: = initial + rate \* 10**



P a g e | **21** LANGUAGE TRANSLATORS

**4. Explain the Cousins of the compiler. (11 marks) (MAY 2012)**

The input to a compiler may be produced by one or more preprocessors, and further processing of the compiler’s output may be needed before running machine code is obtained.

Skeletal source program

↓

**Preprocessor**

↓

Source program

↓

**Compiler**

↓

Target assembly program

↓

**Assembler**

↓

Relocatable machine code

↓

|  |  |  |
| --- | --- | --- |
|  | Library, relocatable object files |  |
| **Loader/ link editor** |  |
|  |  |
|  |  |  |



↓

Absolute machine code

**Preprocessors:**

Preprocessors produce input to compilers. They may perform the following functions:

* Macro Processing:
* File inclusion:
* “Rational” Preprocessors:
* Language extensions:

**Macro Processing:**

* A preprocessor may allow a user to define macros that are shorthand’s for longer constructs.

**File inclusion:**

* A preprocessor may include header files into the program text.
* *For example,* the C preprocessor causes the contents of the file***<global.h>***to replace the statement***#include <global.h>*** when it processes a file containing this statement.

**“Rational” Preprocessors:**

* These processors augment older languages with more modern flow-of-control and data-structuring facilities.
* *For example,* such a preprocessor might provide the user with built-in macros for constructs like whilestatements or if statements.

P a g e | **22** LANGUAGE TRANSLATORS

**Language extensions:**

* These processors attempt to add capabilities to the language by what amounts to built-in macros.
* *For example,* the language Equal is a database query language embedded in C. Statements beginningwith ## are taken by the preprocessor to be database-access statements, unrelated to C, and are translated into procedure calls on routines that perform the database access.

Macro processors deal with two kinds of statements:

* 1. Macro definition and
	2. Macro use
* Definitions are normally indicated by unique character or keyword like define or macro. They consist of a name for the macro being defined and a body, forming its definition.
* The use of macro consists of naming the macro and supply actual parameters, (i.e.) values for its formal parameters.

**Assemblers:**

* Some compilers produce assembly code that is passed to an assembler for further processing.
* Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.
* *Assembly code* is a mnemonic version of machine code, in which names are used instead of binarycodes for operations, and names are also given to memory addresses.
* A typical sequence ***b := a + 2****,* the assembly instructions might be

**MOV a, R1**

**ADD #2, R1**

**MOV R1, b**

* This code moves the contents of the address **a** into register 1, then adds the constant **2** to it, treating the contents of register 1 as a fixed-point number, and finally stores the result in the location named by **b**. thus, it computes b:=a+2.

**Two - Pass Assembly:**

The simplest form of assembler makes two passes over the input, where a *pass* consists of reading an input file once.

* In the *first pass*, all the identifiers that denote storage locations are found and stored in a symbol table.

Identifiers are assigned storage locations as they are encountered for the first time.

**Identifiers** **Address**

1. 0
2. 4

P a g e | **23** LANGUAGE TRANSLATORS

* In the *second pass*, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language and it translates each identifier representing a location into address given for that identifier in the symbol table.
* The output of the second pass is usually *relocatable* machine code, that it can be loaded starting at any location L in memory.

**Loaders and Link-Editors:**

* + Usually, a program called a *loader* performs the two functions of *loading* and *link-editing.*
	+ The process of *loading* consists of taking relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper location.
	+ The *link-editor* allows us to make a single program from several files of relocatable machine code.
	+ These files may be the result of several different compilation and one or more library files of routines provided by the system.
	+ If the files are to be used together, there may be external references, in which the code of one file refers to a location in another file.
1. **Write a short note on grouping of phases? (5 marks)**

In an implementation, activities from more than one phase are often grouped together.

**Front and Back Ends:**

* The phases are collected into a *front end* and a *back end.*
* The ***front end*** consists of those phases that depend primarily on the source language and are largely independent of the target machine.

These normally include

* + lexical analysis
	+ syntactic analysis
	+ semantic analysis
	+ the creating of the symbol table
	+ the generation of intermediate code.
	+ code optimization can be done by the front end as well.
* The front end also includes the error handling that goes along with each of these phases.
* The ***back end*** of compiler includes those portions that depend on the target machine and generally those portions do not depend on the source language, just the intermediate language.

P a g e | **24** LANGUAGE TRANSLATORS

These normally include

* Code optimization
* Code generation
* Error handling and
* Symbol-table operations

**Passes:**

* Several phase of compilation are usually implemented in a single ***pass*** consisting of reading an input file and writing an output file.
* For several phases to be grouped into one pass and for the activity of these phases to be interleaved during the pass.
* For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.
* The token stream after lexical analysis may be translated directly into intermediate code.

**Reducing the Number of Passes:**

* + It is desirable to have relatively few passes, since it takes time to read and write intermediate files. If we group several phases into one pass, it may be forced to keep the entire program in memory, because one phase may need information in a different order then a previous phase produces it.
	+ The grouping into one pass presents few problems.
	+ The interface between lexical and syntax analyzers can often be limited to a single token.
	+ It is very hard to perform code generation until the intermediate representation has been completely generated.
1. **State the different compiler construction tools and their use. (6 marks) (MAY 2013)**

The compiler writer, like any programmer, can profitably use tools such as debuggers, version managers, profilers and so on.

* In addition to these software-development tools, other more specialized tools have been developed for helping implement various phases of a compiler.
* The first compilers were written; systems to help with the compiler-writing process appeared.
* These systems have often been referred to as
	+ ***Compiler-compilers,***
	+ ***Compiler-generators, or***
	+ ***Translator-writing systems.***
* The general tools have been created for the automatic design of specific compiler components.

P a g e | **25** LANGUAGE TRANSLATORS

* These tools use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated.
* 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 a compiler.

The various compiler construction tools are

1. Parser generators
2. Scanner generators
3. Syntax-directed translation engines
4. Automatic code generators
5. Data-flow engines

**Parser generators:**

* These produce syntax analyzers, normally from input that is based on a context-free grammar.
* In early compilers, syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler.
* This phase is considered one of the easiest to implement.

**Scanner generators:**

* These tools automatically generate lexical analyzers, normally from a specification based on regular expressions.
* The basic organization of the resulting lexical analyzer is in effect a finite automaton.

**Syntax directed translation engines:**

* These produce collections of routines that walk the parse tree, generating intermediate code.
* The basic idea is that one or more “translations” are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.

**Automatic code generators:**

* Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.
* The rules must handle the different possible access methods for data.
* Eg; variables may be in registers, in a fixed location in memory or may be allocated a position on a stack. The basic technique is *“****template matching”.***

**Data-flow engines:**

* Much of the information needed to perform good code optimization involves “data-flow analysis,” the gathering of information how values are transmitted from one part of a program to each other part.

P a g e | **26** LANGUAGE TRANSLATORS

1. **Discuss the role of the lexical analyzer. (11 marks) (NOV 2012)**
	* The *lexical analyzer* is the first phase of a compiler.
	* Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses the syntax analysis.
	* This is implemented by making the lexical analyzer be a sub-routine or a co-routine of the parser.
	* Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.



***Interaction of lexical analyzer with parser***

The lexical analyzers is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface.

* One task is stripping out from the source program *comments and white space* in the form of *blank*, *tab and new line characters.*
* Another is *error messages* from the compiler in the source program.

The lexical analyzers are divided into a cascade two phases are

1. *Scanning* is responsible for doing simple tasks.
2. *Lexical analysis* more complex operations

For example, a FORTRAN compiler might use a scanner to eliminate blanks from the input.

**Issues in Lexical Analysis:**

There are several reasons for separating the analysis phase of compiling into lexical analysis and

parsing.

1. Simpler design is the most important consideration.
	* Comments and white space have already been removed by lexical analyzer.
2. Compiler Efficiency is improved.
	* A large amount of time is spent reading the source program and partitioning it into tokens.
	* Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

P a g e | **27** LANGUAGE TRANSLATORS

1. Compiler Portability is enhanced.
	* Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.
	* The representation of a special or non-standard symbol such as ↑ in Pascal can be isolated in the lexical analyzer.

**Tokens, Patterns, and Lexemes:**

* *Tokens*- Sequence of characters that have a collective meaning.
* *Patterns*- There is a set of strings in the input for which the same token is produced as output. This setof strings is described by a rule called a pattern associated with the token.
* *Lexeme*- A sequence of characters in the source program that is matched by the pattern for a token.

|  |  |  |
| --- | --- | --- |
| **Token** | **lexeme** | **patterns** |
|  |  |  |
| const | const | const |
|  |  |  |
| if | if | if |
|  |  |  |
| relation | <, <=, =, < >, >, >= | < or <= or = or < > or >= or > |
|  |  |  |
| id | pi, count, D2 | letter followed by letters and digits |
|  |  |  |
| num | 3.1416, 0, 6.02E23 | any numeric constant |
|  |  |  |
| literal | “core dumped” | any characters between “ and “ except “ |
|  |  |  |

**Attributes for Tokens:**

* When more than one pattern matches a lexeme, the lexical analyzer must provide information about the particular lexeme that matched to the phases of a compiler.
* For example, the pattern **num** matches both the strings 0 and 1.
* The lexical analyzer collects information about tokens into their associated attributes.
* The *tokens* influence *parsing decisions*; the *attributes* influence the *translation of tokens.*
* A token has usually only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token.

The tokens and associated attribute-values for the FORTRAN statement

**E = M \* C \*\* 2**

<id, pointer to symbol-table entry for R> <assign\_op, >

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

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

<num, integer value 2>

P a g e | **28** LANGUAGE TRANSLATORS

**Lexical Errors:**

Possible error recovery actions or Panic mode recovery are

* + 1. Deleting an extraneous character
		2. Inserting a missing character
		3. Replacing an incorrect character by a correct character
		4. Transposing two adjacent characters
1. **Explain the input buffering with sentinels. (6 marks) (NOV 2013)**
	* The two- buffer input scheme is useful when look-ahead on the input is necessary to identify tokens.
	* The techniques for speeding up the lexical analyzer, use the “sentinels “ to mark the buffer end.

The three general approaches to the implementation of a lexical analyzer

1. Use a lexical analyzer generator such as the Lex compiler to produce a regular expression based specification. In this case, the generator provides for reading and buffering the input.
2. Write the lexical analyzer in a conventional systems programming languages using the I/O facilities of that language to read the input.
3. Write the lexical analyzer in assembly language and reading of input.

The lexical analyzer is the only phase of the compiler that reads the source program character-by-character; it is possible to spend a considerable amount of time in the lexical analysis phase.

**Buffer Pairs:**

* Two pointers to the input buffer are maintained.
* The string of characters between the pointers is the current lexeme.
* Initially, both pointers point to the first character of the next lexeme to be found.
* Forward pointer, scans ahead until a match for a pattern is found.
* Once the next lexeme is determined, the forward pointer is set to the character at its right end.
* After the lexeme is processed, both pointers are set to the character immediately past the lexeme.
* The comments and white space can be treated as patterns that yield no token.
* A buffer into two N-character halves, where N is the no.of characters on one disk block, eg. 1024 or 4096.
* If the forward pointer is to move past the halfway mark, the right half is filled with N new input characters.



P a g e | **29** LANGUAGE TRANSLATORS

* If the forward pointer is to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.

**Code to advance forward pointer**

**if** *forward*at the end of first half **then begin** reload second half ;

*forward : = forward + 1;*

**end**

**else if** *forward*at end of second half **then begin** reload first half ;

move *forward* to beginning of first half

**end**

**else** *forward : = forward + 1;*

**Sentinels:**

* The *sentinel* is a special character that cannot be part of the source program.
* Each buffer half to hold a sentinel character at the end (eof).



**Lookahead code with sentinels:** *forward* : = *forward +* 1 ;

**if** *forward*= **eof then begin**

**if** *forward*at end of first half **then begin** reload second half ;

*forward* : = *forward* + 1

**end**

**else if** *forward*at end of second half **then begin** reload first half ;

move *forward* to beginning of first half

**end**

**else / \* eof** within buffer signifying end of input **\* /** terminate lexical analysis

**end**

P a g e | **30** LANGUAGE TRANSLATORS

**9. Explain the specification of tokens? (11 marks)**

Regular expressions are an important notation for specifying lexeme patterns. Each pattern matches a set of strings, so regular expressions will serve as names for set of strings.

1. **Strings and Languages:**
	* The term *alphabet or character class* denotes any finite set of symbols. Typical examples of symbols are letters and characters.
	* The set {0, 1} is the *binary alphabet*.
	* ASCII and EBCDIC are two examples of computer alphabet.
	* A ***string*** over some alphabet is a finite sequence of symbols drawn from that alphabet.
	* The length of string s, usually written |s|, is the number of occurrences of symbols in s.
	* The *empty string* denoted Ɛ, is a special string of length zero.
	* The term ***language*** denotes any set of strings over some fixed alphabet. Abstract languages like , the empty set, or {},the set containing only the empty string, are languages.
	* If x and y are strings, then the *concatenation* of x and y is also string, denoted xy, is the string formed by appending y to x.
	* For example, if x = ban and y = ana, then xy = banana.
	* The empty string Ɛ is the identity element under concatenation; that is, for any string s, SƐ = ƐS= s.
2. **Operations on Languages:**

There are several important operations that can be applied to languages.

In lexical analysis

* Union
* Concatenation
* Closure



P a g e | **31** LANGUAGE TRANSLATORS

**Example:**

* + Let L be the set of letters {A, B, . . . , Z, a, b, . . . , z } and D be the set of digits {0,1,.. .9}.
	+ L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits.
		1. L U D is the set of letters and digits
		2. LD is the set of strings consisting of a letter followed by digit
		3. L4 is the set of all 4-letter strings.
		4. L\* is the set of all strings of letters, including e, the empty string.
		5. L (L U 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.
1. **Regular Expressions:**
	* Regular expression is a notation for describing string. In Pascal, an identifier is a letter followed by zero or more letter or digits.
	* The Pascal identifier as

**letter (letter | digit) \***

The rules is the specification of language denoted by

1.  is a regular expression that denotes {}, the set containing empty string.
2. If ***a*** is a symbol in , then *a* is a regular expression that denotes {*a*}, the set containing the string *a*.
3. Suppose ***r*** and ***s*** are regular expressions denoting the language L(*r*) and L(*s*), then
	1. (*r*) |(*s*) is a regular expression denoting L(*r*)  L(*s*).
	2. (*r*)(*s*) is regular expression denoting L (*r*) L(*s*).
	3. (*r*) \* is a regular expression denoting (L (*r*))\*.
	4. (*r*) is a regular expression denoting L (*r*).

A language denoted by a regular expression is said to be a *regular set.*

Unnecessary parentheses can be avoided in regular expression

1. The unary operator \* has the highest precedence and is left associative.
2. Concatenation has the second highest precedence and is left associative.
3. | has the lowest precedence and is left associative.

P a g e | **32** LANGUAGE TRANSLATORS

1. **Regular Definitions:**
	* For notation, give names to regular expressions and to define regular expressions using these names as if they were 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 each di is a distinct name, and each ri is a regular expression.

**Example:**

1. The set of *Pascal identifier* is the set of strings of letters and digits beginning with a letter. The regular definition is

**letter** A | B | C | … | Z | a | b | … | z **digit** 0 | 1 | 2 | … | 9

**id**  **letter ( letter | digit )\***

1. *Unsigned numbers* in Pascal are strings such as 5280, 39.37, 6.336E4 or 1.894E-4.

The regular definition is

**digit** 0 | 1 | 2 | … | 9 **digits**  **digit digit\***

**optional\_fraction**  **. digits** | **optional\_exponent**( **E** ( + | -|) **digits**) |**num**  **digits optional\_fraction optional\_exponent**

1. **Notational Shorthands:**
	1. **One or more instances**

Unary postfix operator + means “one or more instances of”.

* 1. **Zero or one instance**

Unary postfix operator ? means “ zero or one instances of ”.The regular definition for **num** **digit** 0 | 1 | 2 | … | 9

**digits**  **digit+ optional\_fraction** (. **digits** ) ?

**optional\_exponent** ( E ( + | -) ? **digits)** ?

**num**  **digits optional\_fraction optional\_exponent**

P a g e | **33** LANGUAGE TRANSLATORS

* 1. **Character classes**
		+ The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
		+ The character class such as [a-z] denotes the regular expression a | b | … | z.
		+ Using character classes, we describe identifiers as being strings generated by the regular expression,

**[ A – Z a – z ] [ A – Z a – z 0 – 9 ] \***

1. **Illustrate the steps involved in the recognition of tokens? (11 marks) (NOV 2011)(MAY 2013)**

We considered the problem of how to specify tokens and recognize them.

Consider the following grammar *stmt* **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| 

*expr*  *term* **relop** *term* | *term*

*term*  *id*

| *num*

where the terminals **if, then**, **else**, **relop**, **id**, and **num** generate set of strings given by the following regular definitions:

**if** if **then** then **else** else

**relop** < | <= | > | >= | = | < > **id**  **letter ( letter | digit )\***

**num**  **digit + (. digit + ) ? ( E(+ | -) ? digit + ) ?**

The lexical analyzer will recognize the **keywords** *if, then, else*, as well as the lexemes denoted by ***relop, id***, and ***num.***

We assume lexemes are separated by **white space** consisting of *blanks, tabs, and newlines*. In lexical analyzer will strip out white space.

**delim**  **blank** | **tab** | **newline ws**  **delim +**

If a match for ***ws*** is found, the lexical analyzer does not return a token to the parser.

P a g e | **34** LANGUAGE TRANSLATORS

* To construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute value, using the translation table.
* The attribute values for the relational operators are given by the symbolic constants LT, LE, EQ, NE, GT,GE.



Regular expression pattern for tokens

**Transition diagram:**

* An intermediate step in the construction of a lexical analyzer, produce a stylized flowchart called ***a*** ***transition diagram.***
* Transition diagram depict the actions that take place when a lexical analyzer is called by the parser to get the next token.
* Transition diagram to keep track of information about characters that are seen as the forward pointer scans the input.
* Moving from position to position in the diagram as characters are read.
* Positions in a transition diagram are drawn as circles and are called **states.**
* The states are connected by arrow, called **edges**.
* Edges leaving state **s** have **labels** indicating the input characters that can next appear after the transition diagram has reached state ***s***.
* The **label other** refers to any character that is not indicated by any of the other edges leaving s.
* Transition diagram are **deterministic**, ie no symbol can match the labels of two edges leaving one state.

P a g e | **35** LANGUAGE TRANSLATORS

* One state is labeled as the ***start state***; it is the initial state of the transition diagram where control resides when we begin to recognize a token.
* Certain states may have actions that are executed when the flow of control reaches that state.
* On entering a state we read the next input character.
* If there is an edge from the current state whose label matches this input character, we then go to the state pointed to by the edge.
* Otherwise we indicate failure.

**Transition diagram for relational operators:**



**Transition diagram for identifiers and keywords:**



P a g e | **36** LANGUAGE TRANSLATORS

**Transition diagram for digits:**



**Transition diagram for delim:**



**Implementing a Transition diagram:**

* A sequence of transition diagram can be converted into a program to look for the tokens by the diagrams.
* The systematic approach that works for all transition diagram and constructs programs whose size is proportional to the number of states and edges in the diagrams.

P a g e | **37** LANGUAGE TRANSLATORS

1. **Elaborate on the language for specifying lexical analyzer. (6 marks) (NOV 2013)**
	* Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions.
	* The use of regular expressions for specifying tokens patterns.
	* A particular tool called ***Lex,*** which is used to specify lexical analyzer for a variety of languages.
	* We refer to the tool as the ***Lex compiler*** and to its input specification as the Lex language.

**Creating a lexical analyzer with Lex:**

1. First, a specification of a lexical analyzer is prepared by creating a program ***lex.l*** in the Lex language.
2. Then, ***lex.l*** is run through the ***Lex compiler*** to produce a *C program* ***lex.yy.c.***
3. The program ***lex.yy.c*** consists of tabular representation of a transition diagram constructed from regular expression of ***lex.l***, together with a standard routine that uses the table to recognize lexemes.
4. The actions associated with regular expressions in *lex.l* are pieces of *C code* and are carried over directly to ***lex.yy.c.***
5. Finally ***lex.yy.c*** is run through the ***C compiler*** to produce an object program ***a.out****,* which is the lexical analyzer that transforms an input stream into a sequence of tokens.



**Lex Specifications:**

A Lex program consists of three parts: ***declarations***

***%%***

***translation rules***

***%%***

***auxiliary procedures***

P a g e | **38** LANGUAGE TRANSLATORS

*
* The *declarations section* includes declarations of variables, manifest constants, and regular definitions.
* The *translation rules* of a Lex program are statement of the form

*p*1{ *action1*} *p*2{ *action2*}

…… …..

*p*n{ *actionn*}

where each ***pi*** is a regular expression and each ***actioni*** is a program fragment describing what action the lexical analyzer should take when pattern matches a lexeme*.*

* The *auxiliary procedures* are needed by the actions. These procedures can be compiled separately and loaded with the lexical analyzer.

A lexical analyzer created by Lex behaves with a parser in the following manner.

* When activated by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions ***pi***.
* Then it executes ***actioni.***
* The lexical analyzer returns a single quantity, the token, to the parser.
* To pass an attribute value with the information about the lexeme, we can set a global variable called ***yylval.***

**Lex Program for the tokens:**

%{

/\* ***definitions of manifest constants***

LT, LE, EQ, NE, GT, GE,

IF, THEN, ELSE, ID, NUMBER, RELOP \*/

%}

/\* ***regular definitions*** \*/

delim [ \t\n]

ws {delim}+

letter [A-Za-z]

digit [0-9]

id {letter}({letter}|{digit})\*

number {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

P a g e | **39** LANGUAGE TRANSLATORS

***%%***

{ws} {/\* no action and no return \*/}

if {return(IF);}

then {return(THEN);}

else {return(ELSE);}

{id} {yylval = install\_id(); return(ID); }

{number} {yylval = install\_num(); return(NUMBER);}

“<“ {yylval = LT; return(RELOP); }

“<=“ {yylval = LE; return(RELOP); }

“=“ {yylval = EQ; return(RELOP); }

“<>“ {yylval = NE; return(RELOP); }

“>“ {yylval = GT; return(RELOP); }

“>=“ {yylval = GE; return(RELOP); }

***%%***

install\_id()

{

/\* procedure to install the lexeme, whose first character is pointed to by yytext, and whose length is yyleng, into the symbol table and return a pointer \*/

}

install\_num()

{

/\* similar procedure to install a lexeme that is a number \*/

}

P a g e | **40** LANGUAGE TRANSLATORS

 **IMPORTANT QUESTIONS**

* 1. **MARKS**
1. What is hierarchical analysis? **(NOV 2011) (Ref.Qn.No.8, Pg.no.3)**
2. What is the major advantage of a lexical analyzer generator? **(NOV 2011) (Ref.Qn.No.30, Pg.no.8)**
3. List out the parts on Lex specifications. **(MAY 2012) (Ref.Qn.No.44, Pg.no.10)**
4. What is Compiler? **(MAY 2012) (NOV 2012) (Ref.Qn.No.1, Pg.no.2)**
5. What is transition diagram? **(NOV 2012) (Ref.Qn.No.41, Pg.no.9)**
6. Why separate lexical analysis phase is required? **(MAY 2013) (Ref.Qn.No.29, Pg.no.7)**
7. State the function of front end and back end of a compiler phase. **(MAY 2013) (Ref.Qn.No.22,23, Pg.no.6)**
8. State with example the cousins of compilers. **(NOV 2013) (Ref.Qn.No.17, Pg.no.5)**
9. List the role of lexical analyzer? **(NOV 2013) (Ref.Qn.No.27, Pg.no.7)**

**11 MARKS**

**NOV 2011(REGULAR)**

**1.** Draw the different phases of a compiler and explain. **(Ref.Qn.No.3, Pg.no.16)**

**(OR)**

**2.** How to recognize the tokens? **(Ref.Qn.No.10, Pg.no.34)**

**MAY 2012(ARREAR)**

**1.** Explain the Cousins of the compiler. **(Ref.Qn.No.4, Pg.no.22)**

**(OR)**

**2.** Discuss the Phases of a compiler. **(Ref.Qn.No.3, Pg.no.16)**

**NOV 2012(REGULAR)**

**1.** Explain the phases of a compiler. **(Ref.Qn.No.3, Pg.no.16)**

**(OR)**

**2.** Discuss the role of the lexical analyzer. **(Ref.Qn.No.7, Pg.no.27)**

**MAY 2013(ARREAR)**

**1.** a) State the different compiler construction tools and their use. (6) **(Ref.Qn.No.5, Pg.no.25)** b) Illustrate the steps involved in the recognition of tokens. (5) **(Ref.Qn.No.10, Pg.no.34)**

**(OR)**

**2.** With a neat sketch discuss the functionalities of various phases of a compiler. **(Ref.Qn.No.3, Pg.no.16)**

**NOV 2013 (REGULAR)**

1. Describe the different stage of a compiler with an example. Consider an example for a simple arithmetic expression statement. **(Ref.Qn.No.3, Pg.no.16)**

**(OR)**

1. Explain the buffered I/O with sentinels. Elaborate on the language for specifying lexical analyzer.

**(Ref.Qn.No.8, Pg.no.29) (Ref.Qn.No.11, Pg.no.38)**

P a g e | **41** LANGUAGE TRANSLATORS