B. Tech.

FIFTH SEMESTER EXAMINATION, 2006-07

COMPILER DESIGN

Time : 3 Hours **Total Marks : 100**

Note : (i) Answer *ALL* questions.

(ii) All questions carry equal marks.

(iii) In case of numerical problems assume data wherever not provided.

(iv) Be precise in your answer.

Q. 1. Attempt *any four* parts of the following : **5×4 = 20**

(a) Discuss the role of compiler-writing tools. Describe various compiler writing tools

Ans. For compiler writing basically two tools are used :

(i) LEX : for writing the lexical analyzer.

(i) YACC : for generating a parser with the ability to automatically recover from the errors.

The input for LEX is regular expression specifying the token to be recognized and generates a C program as output that acts as a lexical analyzer for the tokens specified by the inputted regular expression.

When YACC-generated parser encounters an error, it finds the top-most state on its stack, whose underlying set of items includes an item of the from A →errors.

(b) Describe the technique used for reducing number of passes.

Ans. The number of passes, and the grouping of phases into passes, are usually dictated by a variety of considerations germane to a particular language and machine.

Since each phase is a transmission on a stream of data representing an intermediate form of the source program. Several phases can be combined into one pass without reading and writing of intermediate files. In same cases one pass produces its output with little or no memory of prior inputs. In other cases, we may marge phases into one pass by means of a technique known as **back patching.**

(c) Discuss the role of Macros in programming languages.

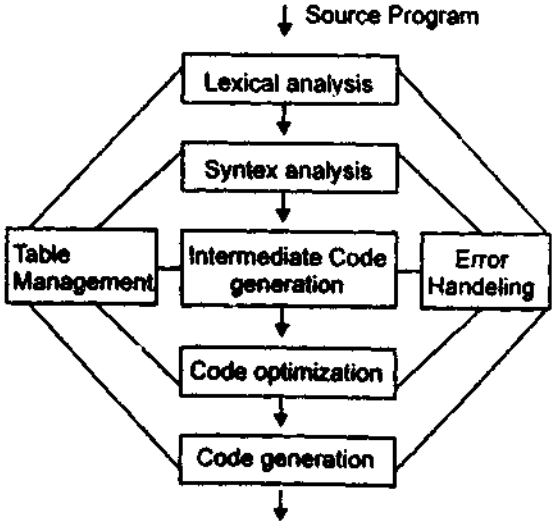
Ans. Many assembly (or programming) languages provide a "macro" facility whereby a macro statement will translate into a sequence of assembly language statements and perhaps other macro statements before being translated into machine code. Thus, a macro facility is a text replacement capability.

(d) Discuss the aspects of high level languages which make them preferable to machine or assembly language.

Ans. A high level programming language makes the programming task simpler. A high level programming language allows a programmer to express algorithms in a more natural notation that avoids many of the details of how a specific computer function.

(e) Describe basic structure of compiler.

Ans. A compiler takes as input a source program and produces as output an equivalent sequence



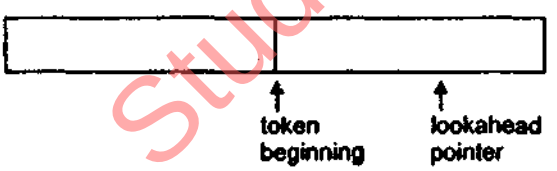
Phases of a compiler

of machine instructions. The process is complex, for this reason, we partition the compilation process into a series of sub processes called "phases." A phase is a logically cohesive operation that takes as input one representation of the source, program and produces as output another representation.

Q. 2. Attempt any two parts of the following : $10 \times 2 = 20$

(a) How lexical analyzer removes white spaces from a source file ? Explain the buffer input scheme for scanning the source program.

Ans. The lexical analyzer scans the characters of the source program one at a time to discover tokens. Often, however, many characters beyond the next token may have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. There are many schemes that can be used to buffer input, and we shall discuss only one here. Figure shows a buffer divided into tow halves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A lookahead pointer scans ahead to the beginning point, until the token is discovered. We view the position of each pointer as being between the character last read and the character next to be read. In practice, each buffering scheme adopts one convention; either a pointer is at the symbol last read or the symbol it is ready to read.



Input buffer

The distance which the lookahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see

DECLARE (ARG1,ARG2, ... , ARG*n*)

without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right-parenthesis. In either case, the token itself ends at the second E. If the lookahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file.

Since the buffer of fig. is of limited size, there is an implited constraint on how much looka head can be used before the next token is discovered. For example, in fig. if the lookahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we choose or use another buffering scheme, we cannot ignore the fact that lookahead is limited.

(b) Develop an algorithm to simulate an NFA reading an input string. What is time and space complexity to your algorithm as a function of size and length of the input string.

Ans. Since an NFA is a finite automata in which there may exist more than one path corresponding to x in Σ^*, and if this is, indeed, the case, then we are required to test the multiple paths corresponding to x in order to decide whether or not x is accepted by the NFA, because, for the NFA to accept x, at least one path corresponding to x is required in the NFA. This path should start in the initial state and end in one of the final states. Whereas in a DFA, since there exists exactly one path corresponding to x in Σ^*, it is enough to test whether or not that path starts in the initial state and ends in one of the final states in order to decide whether x is accepted by the DFA or not.

Therefore, if x is a string made of symbols in Σ of the NFA (*i.e.*, x is in the Σ^*), then x is accepted by the NFA if at least one path exists that corresponds to x in the NFA, which starts in an initial state and ends in one of the final states of the NFA. Since x is a member of Σ^* and there may exist zero, one, or more transitions from a state on an input symbol, we define a new transition function, δ_1, which defines a mapping from $2^Q \times \Sigma^*$ to 2^Q; and if $\delta_1(\{q_0\}, x) = P$, where P is a set containing at least one member of F, then x is accepted by the NFA. If x is written as wa, where a is

the a last symbol of x, and w is a string made of the remaining symbols of x then :

$\delta_1(\{q_0\},x) = \delta_1(\delta_1\{q_0\},w)a)$ since δ_1 defines a mapping from $2^Q \times \Sigma^*$ to 2^Q

$\delta_1(p,a) = \cup_{\text{for every q in P}}\ \delta(q,a)$

For example, consider the finite automata shown below :

$M = (\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_3\})$

where :

$\delta(q_0,0) = \{q_1\}$, $\quad \delta(q_0,1) = \Phi$

$\delta(q_1,0) = \{q_1\}$, $\quad \delta(q_1,1) = \{q_1, q_2\}$

$\delta(q_2,0) = \Phi$, $\quad \delta(q_2,1) = \{q_3\}$

$\delta(q_3,0) = \{q_3\}$ $\quad \delta(q_3,1) = \{q_3\}$

If $x = 0111$, then to find out whether or not x is accepted by the NFA, we proceed as follows :

$\delta_1(\{q_0\},0) = \delta(q_0,0) = \{q_1\}$

Therefore $\delta_1(\{q_0\},01) = \delta_1(\delta_1(\{q_0\},0),1)$

$= \delta_1(\{q_1\},1) = \delta(q_1,1)$

$= \{q_1, q_2\}$

Therefore

$\delta_1(\{q_0\},011) = \delta_1(\delta_1(\{q_0\},01),1)$

$= \delta_1(\{q_1, q_2\},1)$

$= \delta(q_1,1) \cup \delta(q_2,1)$

$= \{q_1, q_2\} \cup \{q_3\}$

$= \{q_1, q_2, q_3\}$

Therefore

$\delta_1(\{q_0\},0111) = \delta_1(\delta_1(\{q_0\},011),1)$

$= \delta_1(\{q_1, q_2, q_3\},1)$

$= \delta(q_1,1) \cup \delta(q_2,1) \cup \delta(q_3,1)$

$= \{q_1, q_2\} \cup \{q_3\} \cup \{q_3\}$

$= \{q_1, q_2, q_3\}$

Since $\delta_1(\{q_0\},0111) = \{q_1, q_2, q_3\}$, which contains q_3, a member of F of the NFA—, hence $x = 0111$ is accepted by the NFA

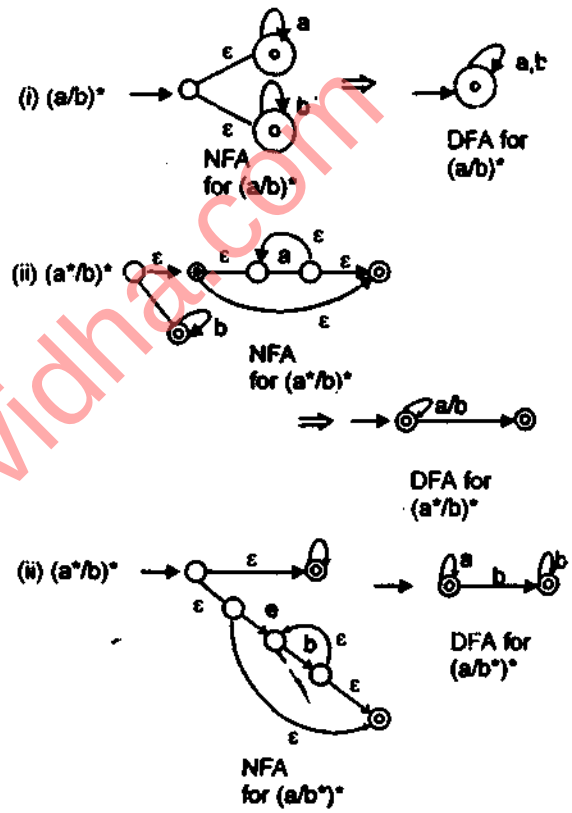
Therefore, if M is a NFA, then the language accepted by NFA is defined as :

$L(M) = \{x | \delta_1(\{q_0\}x) = P$, where P contains at least one member of $F\}$

(c) How is finite automata useful for lexical analysis ? Show that the following regular expressions are same by constructing optimized DFA :

(i) $(a/b)^*$ (iii) $(a^*/b)^*$ (iii) $(a/b^*)^*$

Ans.



Q. 3. Attempt any two parts of the following : 10×2 = 20

(a) What do you understand by left factoring and how it is eliminated ?

Ans. Often the grammar one writes down is not suitable for recursive-descent parsing, even if there is no left-recursion. For example, if we have the two productions.

Statement → if condition then statement else statement

| if condition then statement

we could not, on seeing input symbol if, tell which to choose to expand statement. A useful method for manipulating grammars into a form suitable for recursive-descent parsing is left-factoring, the process of factoring out the common prefixes of alternates.

If $A \to \alpha\beta \mid \alpha\gamma$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta$ or to $\alpha\gamma$. We may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β or to γ. That is, left-factored, the original productions become

$$A \to \alpha A'$$
$$A' \to \beta \mid \gamma$$

(b) Consider the following grammar :

$$E \to T + E \mid T$$
$$T \to V^* T \mid V$$
$$V \to id$$

Write down the procedures for the non terminals of the grammar to make a recursive descent parser.

Ans. In this question we have to remove left recursion. Given grammar is

$$E \to T + E \mid T$$
$$T \to V^* T \mid V$$
$$V \to id$$

eliminating the immediate left-recursion (productions of the form $A \to A\alpha$) we obtain

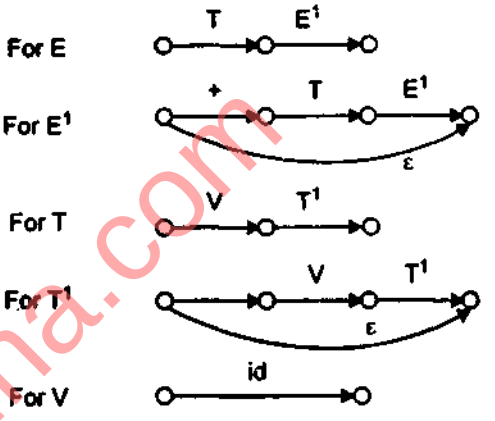
$$E \to TE'$$
$$E' \to *TE' / \varepsilon$$
$$T \to VT'$$
$$T' \to *VT' / \varepsilon$$
$$V \to id$$

There is no guarantee that one can correctly chose a path through a transition diagram of the grammar, although if we eliminate left-recurmion and then left featuring, we have a fair chance of success if we do the following for each non terminal A :

(1) Create an initial and final (return) value.

(2) For each production $A \to X_1 X_2 .. X_n$, create a path from the initial to the final state, with edges labelled $X_1, X_2, ... X_n$

Transition diagram for given Grammer



(c) Discuss the role of data flow analysis

Ans. A number of optimizations can be achieved by knowing various pieces of information that can be obtained only by examining the entire program. For example if a variable A has value 3 every time control reaches a certain point p, then we can substitute 3 for each use of A at p. Knowing that the value of A is 3 at p may require examination of the entire program.

The data flor analysis can be used to gather such information.

Data flow analysis determines information regarding the data flow in a program like how data items are assigned and referenced in a program, what are the values which are available when program execution reaches a specific statement of the program.

Data flow analysis is basically a process of computing the values of a set of items of data flow information which is useful for the purpose of optimization.

Q. 4. Attempt any two parts of the following : $10 \times 2 = 20$

(a) Translate the following program fragment into three address code :

int i;

i = 1

while a < 10 do

if x > y then $a = x + y$

else $a = x - y$

Ans. int i;

i = 1

while a < 10 do

if x > y then $a = x + y$

else $a = x - y$

Three address code :

(1) int i

(2) i = 1

(3) while a < 10 goto (5)

(4) goto (12)

(5) if x > y goto (7)

(6) goto (10)

(7) $t_1 = b + c$

(8) $a = t_1$

(9) goto S. next (if S is given expression)

(10) $t_2 = x - y$

(11) $a = t_2$

(12) exit

(b) Construct LL(1) parsing table for the following grammar :

$S \rightarrow A$

$A \rightarrow aB \mid aC \mid Ad \mid Ae$

$B \rightarrow bBc \mid f$

$C \rightarrow g$

Ans. Given grammar $S \rightarrow A$

$A \rightarrow aB \mid aC \mid Ad \mid Ae$

$B \rightarrow bBc \mid f$

$C \rightarrow g$

First (S) = First (A) = {a}

First (B) = {b, f}

First (C) = {g}

Since the grammar is ε-free, follow sets are not required to be computed in order to enter the productions into the parsing table. Therefore the parsing table is :

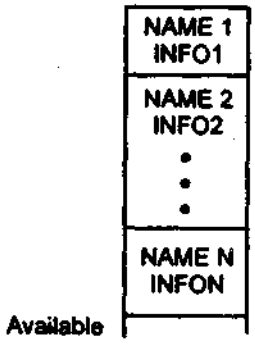
Parsing Table

	a	b	f	g	d
S	S→A				
A	A→aS				A→d
B		B→bBC	B→f		
C				C→g	

(c) Discuss the important data structures which are used in implementing symbol table.

Ans. For implementing symbol table we use following data types :

(1) LIST : The conceptually simplest and easiest-to-implement data structure for a symbol table. We use single array, or equivalently several arrays to store names and their associated informatiens. To, retrive information about a name, we search from the

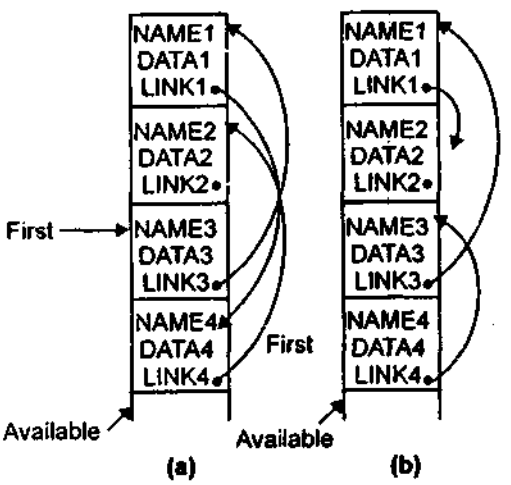


A Linear List of records

beginning of the array upto the position marked by pointer AVAILABLE.

If the symbol table contains n names, the work necessary to insert a new name is proportional to n. To find data about a name we shall, on the average, search n/2 names, so the cost of an enquiry is also proportion to n.

(2) Self Organizing Lists : At the cost of a little extra space we can use a trick that will often save a substantial fraction of the time spent in searching the symbol table. We add a LINK field to each record, and we search the list in the order indicated by the



Self-organizing symbol table

LINK's. Figure (a) shows an example of a four-name symbol table. FIRST gives the position of the first record on the linked list, and each LINK field indicated the next record on the list. In fig. (a) the order is NAME3, NAME1, NAME4, NAME2.

When a name is referenced or its record is first created, we move the record for that name to the front of the list by moving pointers. Figure (b) shows the effect of moving NAME4 to the front of the list. In general, if we search for the find NAME*i* we remember the previous name on the list, say NAME*p*. We temporarily remove entry *i* from the list by making LINK*p* point where LINK*i* points. Then we make LINK*i* point where FIRST points, and finally, we move NAME*i* to the front of the list by making FIRST point to NAME*i*.

(3) Search Trees : A more efficient approach to symbol-table organization is to add two link fields, LEFT and RIGHT, to each record. We use these fields to link the records into binary search tree. This tree has the property that all names NAME*j* accessible from NAME*i* by following the link LEFT*i* and then following any sequence of links will precede NAME*i* in alphabetical order (symbolically, NAME*i* < NAME*i*). Similarly, all names NAME*k* accessible starting with RIGHT*i* will have the property that NAME*i* < NAME*k*. Thus if we are searching for NAME and have found the record for NAME*i*, we need only follow LEFT*i* if NAME < NAME*i* and need only follow RIGHT*i* if NAME*i* < NAME. Of course, if NAME = NAME*i* we have found what we are looking for fig. gives an algorithm to look for NAME in a binary search tree, where P is initially a pointer to the root.

(1) while $P \neq$ null do

(2) if NAME = NAME(P) then .../*NAME found, take action on success*/

(3) else if NAME < NAME(P) then P : = LEFT(P)

/*visit left child */

(4) else /*NAME(P) < NAME */P : = RIGHT(P)

/* visit right chile */

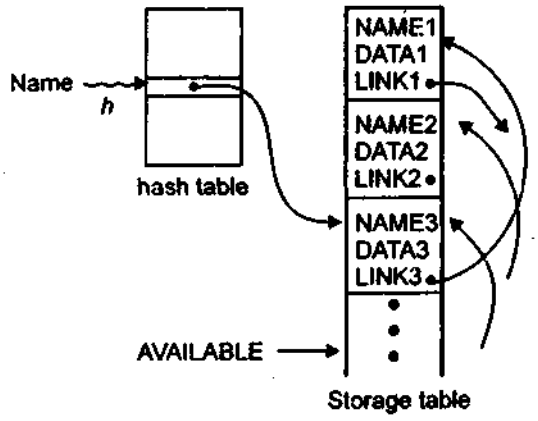
/*if we fall through the loop, we have failed to find NAME*/

Fig. Binary tree search routine

If NAME is found at line (2), P points to NAME' record. If we fall through the loop, we have failed to find NAME. Should we wish to insert a new record for a new name, we check before assigning to P in lines (3) or (4) that LEFT(P) or RIGHT (P), respectively, is not null. If it is null, then P points to a record whose left (resp. right) child should be the record for the new name.

If names are encountered in a random order, the average length of a path in the tree will be proportional to log *n*, where *n* is the number of names. Since each search follows one path from the root, the expected time needed to enter *n* names and make *m* inquiries is proportional to $(n + m) \log n$. If *n* is greater than about 50, there are clear advantages to the binary search tree over the linear list and probably over the linked self-organizing list. If efficiency is paramount, however, there is an even better method than the binary search tree, the hash table

(4) Hash Tables : Many variations of the important searching technique known as hashing have been implemented in compilers. Here we shall consider a rather simple variant. Even this scheme gives us the capability of performing *m* accesses on *n*



Hash table

names in time proportional to $n(n+m)/k$, for any constant k of our choosing. Since k can be made as large as we like, this method is generally superior to linear lists or search trees and is the method of choice for symbol tables in most situations, especially if storage is not particularly costly.

The basic hashing scheme is illustrated in Fig. Two tables, a hash table and a storage table, are used. The hash table consists of k words, numbered $0, 1, \ldots, k-1$. These words are pointers into the storage table to the heads of k separate linked lists(some lists may be empty). Each record in the symbol table appears on exactly one of these lists.

Q. 5. Write short notes on any two of the following sections : 10 ×2 = 20

(a) Principal sources of optimization

Ans. Principle sources of optimization : Code optimization techniques are generally applied after syntax analysis, usually both before and during code generation. The technique consists of detecting patterns in the program and replacing these patterns by equivalent but more efficient constructs. These patterns may be local or global, and the replacement strategy may be machine dependent or machine-independent.

(b) Problems in code generation.

Ans. Problems in Code Generation : It might appear that the task of code generation is now relatively easy. However, difficulties arise in attempting to perform the computation represented by the intermediate-language program efficiently, using the available instructions of the target machine. There are three main sources of difficulty : deciding what machine instructions to generate, deciding in what order the computations should be done, and deciding which registers to use.

What Instructions Should We Generate ?

Most machines permit certain computations to be done in a variety of ways. For example, if our target machine has an "add-one-to-storage" instruction (AOS),then for the three-address statement A : = A + 1 we might generate the single instruction AOS A, rather than the more obvious sequence

```
LOAD A
ADD #1
STORE A
```

Deciding which machine code sequence is best for a given three-address construct may require extensive knowledge about the context in which that construct appears. We shall have more to say on this matter when we discuss the choice of registers.

In What Order Should We Perform Computations ?

The second source of difficulty concerns the order in which computations should be performed. Some commutation orders require fewer registers to hold intermediate results than others. Picking the best order is a very difficult problem in general. Initially, we shall generate code for the three-address statements in the order in which they have been produced by the semantic routines.

What Registers Should We Use ?

The final problem that we shall mention is register assignment, that is, deciding in which register each computation should be done. Deciding the optimal assignment of registers to variables is difficult, even with single-register quantities. The problem is further complicated because certain machines require register-pair (an even and next odd-numbered register) for some operands and results.

For example, in the IBM System/370 machines, integer multiplication and integer division

involve register pairs. The multiplication instruction is of the form M X, Y

where X, the multiplicand, refers to the even register of an even/odd register pair. The multiplicand itself is taken from the odd register of the pair. Y represents the multiplier. The product occupies the entire even/odd register pair.

Division instruction is of the form

D X, Y

where the 64-bit dividend occupies an even/ odd register pair whose even register is X. Y represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

(c) Error recovery schemes.

Ans. Panic Mode : Since the details of the recovery methods vary somewhat depending on the type of parsing technique used, we shall examine methods for recovering from synthectic errors using operator precedence, LL and LR Parsers. Before doing so we might mention a crude but effective systematic method for error recovery in any kind of parsing, the so-caleld 'panic mode' of recovery.

In panic mode a parser discards input symbols (untill a 'synchronizing' token, usually a statement delimiter such as semicolon or end, is encountered. The parser then detects stack entries until it finds an entry such that it can continue parsing, given the synchronizing token can the input. Two virtues of this recovery method are that is simple to implement and unlike some insertion schemes, it can never get into an infinite loop.