B.E.

Seventh Semester Examination, December-2008

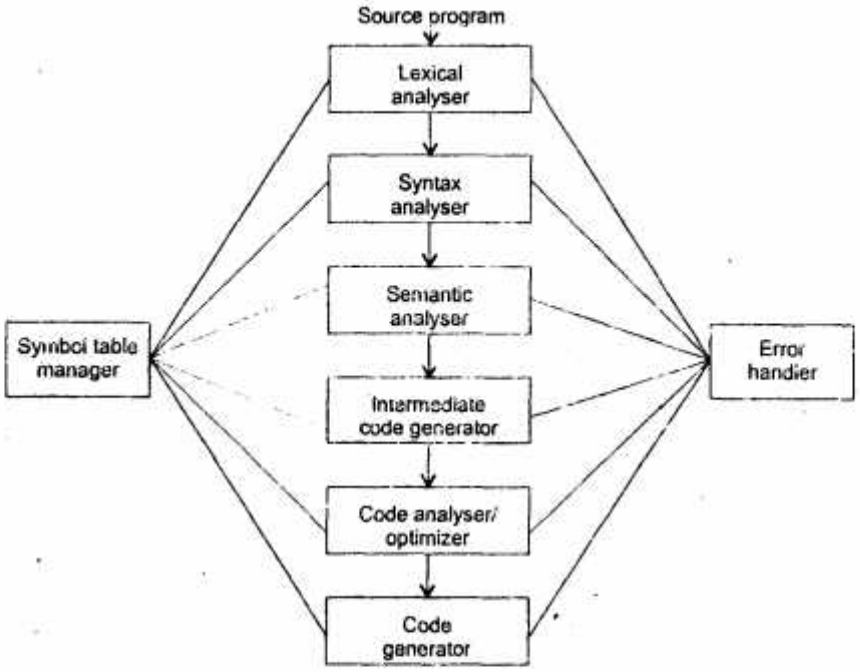
Compiler Design (CSE-405-E)

Note : Attempt any five questions.

Q. 1. (a) Describe the structure of compiler with its phases with the help of an example.

Ans. Conceptually a compiler operates in phases, each of which transforms the source program from one representation to another.

A typical decomposition of a compiler is shown in fig. In practice, some of the phases may be grouped together, & the implementially intermediate representations between the grouped phases need not be explicitly constructed.



Phases of a comilar

Q. 1. (b) Which translator is better single pass or multi pass and why ?

Ans. A translation scheme is a contect free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of productions. In the simple translation schemes, the attributes were of string type, one for each symbol and for every production $A \rightarrow \lambda_1 \ldots\ldots x_n$, the semantic rule formed the string for A by concatenation the strings for $x_1 \ldots\ldots x_n$, in order with some optional additional strings in between.

Q. 2. (a) Define following term Lex, Lexene, Lexical analyzed and token. Describe the role of a lexical analyzer.

Ans. Lexical Analysis : In a compiler, linear analyses is called a scanning. For example, in lexical analysis the characters in the assignment statement.

Position = initial + rate * 60

Would be grouped into following tokens :

(i) The identifier position.

(ii) The assignment symbol.

(iii) The identifier initial

(iv) The plus sign

(v) The identifier rate

(vi) The multiplication sign.

(vii) The number 60.

Lex : Patterns are specified by regular expressions & a compiler for lex can generate an efficient finite automation recognizer for the regular expressions.

Q. 2. (b) Convert the regular expression (a+b)* abb into e-NFA and then corresponding DFA.

Ans. eNFA :

S → aABc
A → Abc / b
B → d
abbcdc
aAbcde
aAdc
aABe
S ⇒ aABc ⇒ aAde ⇒ aAbcde ⇒ abbcde
rm rm rm rm

DFA :

S : = F_closure ({S_0});
a : = nextchar;
while $a \neq$ cof do begin
S : = E_closure (more (s, a)),
a : = nextchar;
end
if $S \cap F \neq \phi$ then
return "yes";
else
return "no";

Q. 3. (a) What is a passer ? Write the predictive passing algorithm with example.

Ans. Passer Generator : These produce syntax analysers, 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 intelligential effort of working a compiler.

Predictive Passing : Recursive-descent passing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. Here we consider a special form in which the look ahead symbol unambiguously determines the procedure selected for each non-terminal.

Example :

```
procedure match (t : token);
begin
    if look ahead = t then
    look a head : = next token
    else error
end;
```

Q. 3. (b) What is left recursion and left factoring ? How these can be removed ?

Ans. It is possible for a recursive descent to loop forever. A problem arises with left recursive productions line

$$\text{expr} \rightarrow \text{expr} * \text{terms}$$

A left recursive production can be eliminated by re-writing the offending production. Consider a non termian A with 2 productions.

$$A \rightarrow A\alpha | B$$

Where α & β are sequence of terminals & non-terminals that donot start with A.

Example : expr → expr + term | term

Left factoring : LF is a grammar transformation that's useful for producing a grammar suitable for predictive passing. The basic idea is that when it is not clear which of the 2 alternative productions to use, to expand a non-terminal A, the way be able to rewrite the A, productions to defer the decision until we have seen enough of input to make the right choice.

Example : stmt → if, expr then stmt else stmt

if expr then stmt

Q. 4. (a) What is ambiguity ? How an ambiguous grammar is converted to unambiguous grammar ? Explain with the help of an example.

Ans. We have to be careful in talking about the structure of a string according to a grammar. While it is clear that each parse tree derives exactly the string read off its leaves, a grammar can have more than one parse tree generating a given string of token. Such a grammar is said to be ambiguous.

Example : String → string + string | string – string |0|1|2|3|4|5|6|7|8|9| merging the notation of digit & list into the non-terminal string makes superficial sense, because a single digit is a special case of a list.

Q. 4. (b) What is Handle pruning ? In which passer it is used and how ?

Ans. A rightmost derivation in reverse can be obtained by "handle pruning". That is we start with a string of terminals w that we wish to phrase. If w is a sentence of a grammar at hand, then $w = y_n$, where y_n is the *nth* right-sequential form of some as yet unknown rightmost derivation.

$$S = y_0 \underset{rm}{\Rightarrow} y_1 \underset{rm}{\Rightarrow} \ldots\ldots\ldots\ldots \underset{rm}{\Rightarrow} y_{n-1} \underset{rm}{\Rightarrow} y_n = w$$

Right Sequential Form	Handle	Reducing production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Q. 5. (a) Write the 3-address code, quadruple, triple and indirect triple for the expression.
$-(a+b)*(c-d*e)$

Ans.

	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2i)	uminus	c	
(3i)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

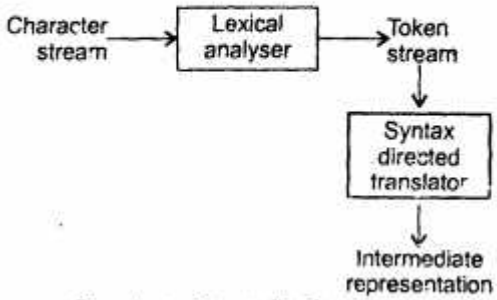
(a) Triples

	op	arg 1	arg 2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3i)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Quadruples

Q. 5. (b) How syntax directed translation is implemented ?

Ans. Besides specifying the syntax of a language, a context free grammar can be used to help guide the translation of program. A grammar oriented compiling technique, known as syntax -directed translation, is very helpful for organising a compiler end front & will be used extensively throughout the unit.



Structure of compiler frame end

Q. 6. (a) What is a symbol table ? Explain the use of hash table for the symbol tables.

Ans. An essential data structure containing a record for each identifier with fields for the attributes of the identifier is called symbol table.

The identifier 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 & rate are seen by the lexical analyser.

The remaining phases enter information about the identifiers into the symbol table & then use this information in various ways.

Symbol table :

1.	Position	.....
2.	Initial	.....
3.	Rate	.....
4.		

Q. 6. (b) What are semantic errors ? Explain with the help of an example.

Ans. We know that programs can contain errors at many different levels. For example, errors can be

(i) Lexical

(ii) Syntactic

(iii) Semantic

(v) Logical

The error handler in a passer has simple to state goals :

(i) It should report the presence of errors clearly & accurately.

(ii) It should not significantly show down the processing of correct programs.

Example :

(i) program prmax (input, output);

(ii) var

(iii) x, y; integer;

(iv) function max (i:integer; j:integer); integer;

(v) {return maximum of integers i & j}

(vi) begin

(vii) if i>j then max: = i

(viii) else max: = j

(ix) end;

(x) begin

(xi) redln (x,y);

(xii) writeln (max (x,y));

(xiii) end

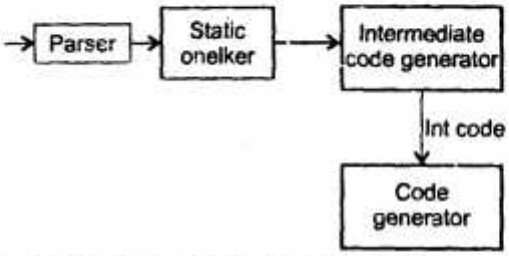
Q. 7. (a) What is code optimization ? What are its various type and stages ? Explain in detail.

Ans. The code optimization phase attention to improve the intermediate code. So that faster running machine code will result.

Some optimizations are trival. For example, a natural algorithm generates the intermediate code (1.3), 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 : = *id*3× 60.0

id1 : = *id*2+ temp 1



Position of intermediate code generator

Q. 7. (b) Describe the concept of register allocation.

Ans. The use of register is often subdivided into 2 subproblems :

(i) During register allocation, we select the set of variables that will reside in registers at a point in the program.

(ii) During a subsequent register assignment phase, we pick the specific register that a variable will reside in

t : = a + b	t : = a + b
t : = t*c	t : = t + c
t : = t/d	t : = t/d
(a)	(b)

(a), (b) two three address code sequences

L	R_1, a	L	R_0, a
A	R_2, b	A	R_0, b
M	R_0, c	A	R_0, c
O	R_0, d	SRDA	$R_0, 3_2$
ST	R_1, t	ST	R_i, d
(c)		(d)	

(c), (d) : Optimal machine code sequences

Q. 8. Write short note on the following :

(a) Labelling algorithm **(b) Global data flow analysis**

(c) Backpatching **(d) Canonical LR Parser**

Ans. (a) Labelling Algorithm : The most elementary programming language construct for changing the flow of control in a program is label and logo.

The stack machine executes instructions in numerical sequence unless told to do otherwise by a conditional or unconditional jump statements. Several options exist for specifying the target of jumps :

(i) The instruction operand gives the target location.

(ii) The instruction operad specifies the relative distance, positive or negative to be jumped.

(iii) The target is specified symbolically i.e., the machine supports labels.

(b) Global Data Flow Aanalysis :

B_5

t_6	:=	$4 * i$
x	:=	$a[t_6]$
t_7	:=	$4 * i$
t_8	:=	$4 * j$
t_9	:=	$a[t_8]$
$a[t_1]$	:=	t_9
t_{10}	:=	$4 * j$
$a[t_{10}]$	:=	x
go to B_2	:=	B_2

(a) Before

B_5

t_6	:=	$4 * i$
x	:=	$a[t_6]$
t_8	:=	$4 * j$
t_9	:=	$a[t_8]$
$a[t_6]$	:=	t_9
$a[t_8]$	:=	x
go to B_2		

(b) After

(c) Backpatching : The easier way to implement the syntax-directed definitions is to use 2 passes.

We can get around this problem by generating a series of branching statements with the targets of the jumps temporarily left unspecified. Each such statement will be put on a list of go to statements whose labels will be filled in when the proper labcl can be determined. We can this subsequent filling in of labels backpatching

Boolean expression :

(i) $E \rightarrow E_1$ or ME_2

(ii) $E \rightarrow E_1$ and ME_2

(iii) $E \rightarrow$ not E_1

(iv) $E \rightarrow (E_1)$

(v) $E \rightarrow id_1$ relop id_2

(vi) $E \rightarrow$ true

(vii) $E \rightarrow$ false

(viii) $M \rightarrow \varepsilon$

(d) Canonical LR Parser : Recall that in the SLR method, state i calls for reduction by $A \rightarrow \alpha$ if the set of items T_i contain item $[A \rightarrow \alpha]$ & a is in follow (A). In some situations, howevers, when state i appears on top of the stack, the variable prefix $\beta\alpha$ on the stack is such that βA can not be followed by 'a' in a right sequential form.

Thus, the reduction by $A \rightarrow \alpha$ would be invalid as input a.

Example :

```
Function closure (I);
begin
repeat
for each item [A→ α. Bβ, a] in I
each production B→ γ in G'
and each terminal b in first (βa)
such that [B→ γ b] is not in I do
add [B → γ b] to I
until no more item can be added to I;
return I
end;
```