* Unit - 4 *

16/12/23

part 1 : Run-time Environment :-

Symbol table, Storage allocation strategies, parameter passing, Source language issues (Activation Record, procedures).

* Symbol table :-

Semantics - meaning of syntax.

It is a data structure used by the compiler to keep track of the semantics of the variable.

* It stores information about indentifiers (name of the variable)
* It stores information about scope and binding information about identifiers.

static (compile-time) dynamic (Run-time)

* Symbol table is built in lexical & syntax analysis phase.

Symbol table used by various phases :-

1) lexical Analysis :- lexical analysis uses symbol table for storing the information of symbols.
2) Syntax Analysis :- verifies the symbol table about the tokens being generated.
3) Semantic Analysis :- Refer symbol table for type conflict issues
4) ICG :- uses symbol table for indentifier information.
5) code generation :- It refers symbol table to know how much run-time space is allocated.

* Entries in Symbol table :-

To achieve compile-time efficiency compiler uses symbol table.

* It associates lexical names with attributes
  => Attributes specifies information about the scope, type and storage of the variable.

Name	Attributes

* We store names in 2 ways : 1) Fixed length name
in Symbol-tables 2) variable length name.

-How to store names in Symbol tables :-

* Fixed length name :- A fixed space for each name is allocated in Symbol table.
-> In this type of storage if name is too small, then there is a wastage of space.

* Variable length name :- The amount of space required by the string is used to store the name.
=> the name can be stored with the help of starting index and the length of each name.

Name		Attributes
starting index	length	
0	9	

* Items to be stored in Symbol table :-

1) variable names
2) constants
3) procedure names
4) Function names.

* uses of Symbol table :-

*) used in quick searching and insertion of an indentifier.

* Storage allocation strategies :-

1) Static Allocation :- Memory is created for the program only at the compile time. and deallocated after program completion.
* It is similar to static memory allocation in c language.

* Memory is created in static area and memory is allocated only one time.

* Drawbacks of Static memory Allocation :-

1) we must know in advance the size of the array i.e the size is fixed.

2) It doesn't support recursion.

2) Stack Allocation :-

Stack is a data structure which works on the principle of LIFO. when a procedure or function call occurs, then an activation record is created for the corresponding function.

* And the activation record is pushed on to the top of the stack. At the end it is popped from the stack.

* Activation record is a contigious block of storage and manages the information required by the single execution of procedure.

* It supports recursion.

3) Heap Allocation :-

It is more flexible allocation. Allocation and deallocation of memory to user can be done at anytime.

* It supports recursion process.

* It overcomes the problem of static memory allocation.

* It is used to allocate the memory for variables dynamically.

* parameter passing :-

1) Formal parameters :- parameters within called function.

2) Actual parameters :- parameters within calling function.

l-value → a = 3*4+5

(contains) stores ⇓

variable which address of the r-value

r-value → 17

⇓

single value that appears on the right side of the assignment operator

* parameter passing techniques :-

1) call by value
2) call by reference
3) call by copy & Restore
4) call by Name.

1) call by value :- In this, the values are passed from calling function (actual parameters) to formal parameters (called function)
   * Here we are passing 'r' value.
   * Any changes done at formal parameters will not effect formal parameter.

2) call by reference :- Reference means address.
   * Here we are passing the 'L' value.
   * Any changes in formal parameters will have effect on formal parameters.

3) call by copy & Restore :- call by copy means passing the values from actual to formal parameters.
   * In this we are sending 'r' values.
   * And restore means passing the values from formal to actual.
   * Here we are updating the 'L' value.

4) call by Name :- All the names of formal parameters are replaced with the names of actual parameters.
   * This is known as lazy evaluation.

Ex:-

```
main ()
{
  int a=3, b=4;
  Pf (a,b);
  swap (a,b);
  Pf (a,b);
}
```

actual parameters

r-values

```
swap (int c, int d)
{
  int t;
  t = c;
  c = d;
  d = t;
}
```

formal parameters.

Actual parameters changes :-

1) call by value - 3, 4

2) call by reference - 4, 3

a [3]	b [4]	c [4]	d [3]
1000	2000	1000	2000

swap (4a, 4b)

swap (int *c, int * d)

3) call by copy & restore - 4, 3

A.P - F.P - 3, 4

F.P - A.P - 4, 3

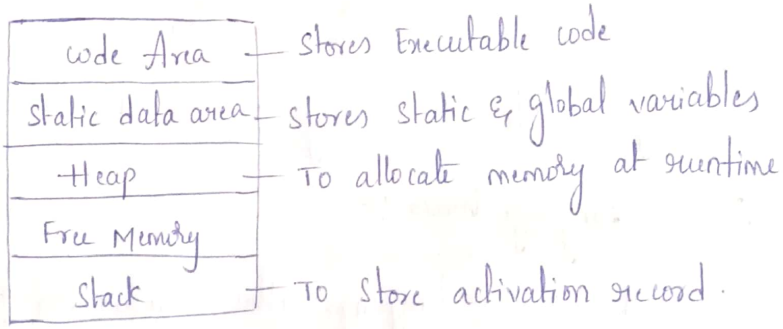
4) call by Name - 3, 4.

* Storage Organization :-

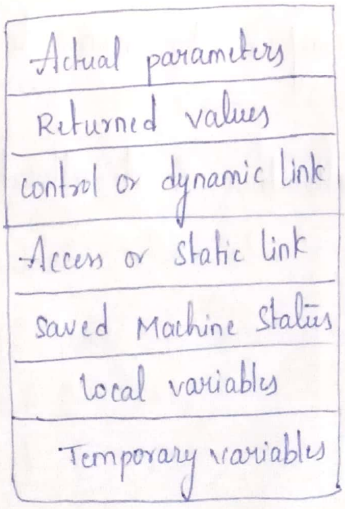
1) Runtime Environment :- If we take any program and save the program then the program will be saved in harddesk.

And during compilation also program will store in hard-desk only. But, CPU will run the program when it is in main memory. so, the compiler demands OS to allocate some block of memory in order to store the program in main memory.

=>



Fields in Activation Record :-

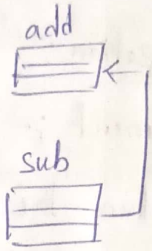


* The parameters of calling function are known as Actual parameters
* Returned values :- To store the results of function call, we use returned values.

Ex:-
```
a = sum();   - func call
int sum()
{
  - - -
  - - -
}          - func dynition.
```
a = returned int value.

* control or dynamic link :- It points the activation record of calling function.

Ex:-
```
add()
{
  sub()
}
```



* Access or static link :- It refers to the local data of called function, but found in another activation record.
* saved Machine status :- It stores the address of next instruction to be executed.
* Local variables :- Local variables are stored inside the functio
* Temporary variables :- They are used to evaluate some expressio
* Source Language issues :-

1) Does the source language allow recursion ?
2) How the parameters are passed to the procedure ?
3) Does the procedure refers to non-local names and how ?
4) Does the language allocation & de-allocation dynamically ?

* Procedure :- A program is made up of procedure. It is a declaration that associates an identifier with a statement (name and body).

* Activation Tree :-

Activation :- procedure under execution.

Activation record :- The information needed to execute is stored in contigious memory.

* Activation tree :- Flow of execution of a program. It represents the control flow of program in graphical manner. It is used in managing a stack of activation records.
* Activation tree is a way how control enters and exit the procedure.
* It is represented as a node.
* If a and b are 2 procedures, their activations will be non-overlapping when one is called after another procedure.
* Overlapping are nested procedures.
* The root represents the activation of the main program.
* The node of 'a' is a parent for node 'b' if and only if control flows from a to b.
* The node for 'a' is to the left of node 'b' if only if the lifetime of a occurs before the lifetime of b.

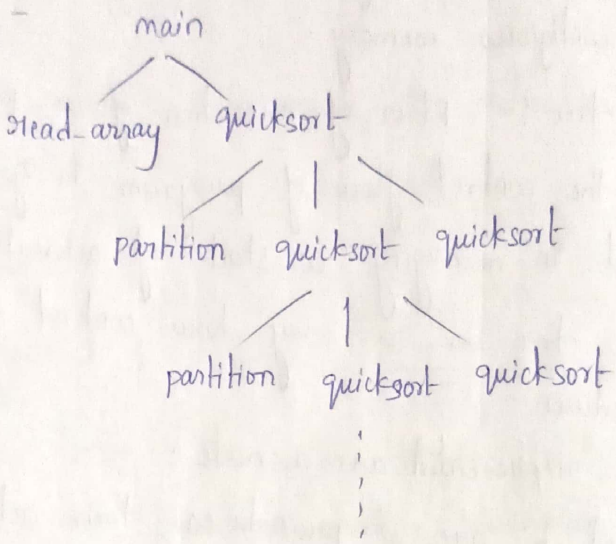
Ex:-

```
main()
{
  int n;
  read_array();   } non-overlapping procedures
  quicksort(1,n); } Actual parameters
}
void quicksort(int m, int n)  ---- procedure name
{
  int i = partition(m,n);   } Body of the
  quicksort(m, i-1);        } procedure
  quicksort(i+1, n);
}
```

nested procedure

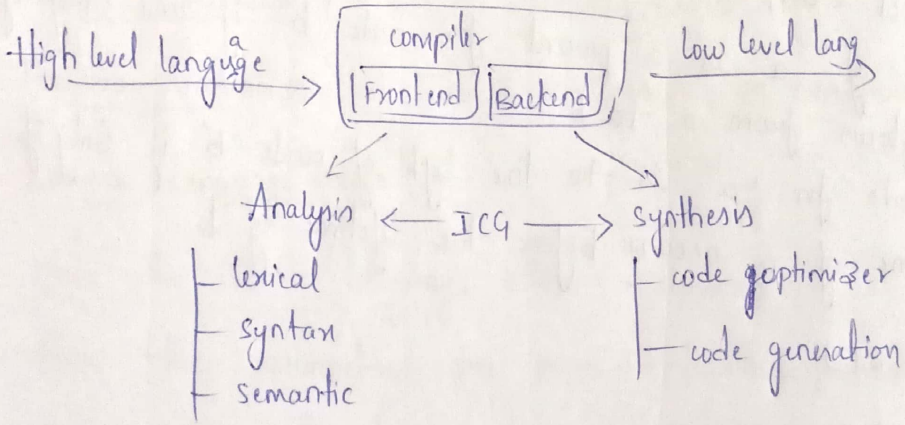
```
void read_array ()
{
  - - -
  - - -
}
int partition (int y, int z)
{
  - - -
  - - -
}
```

Activation tree :-

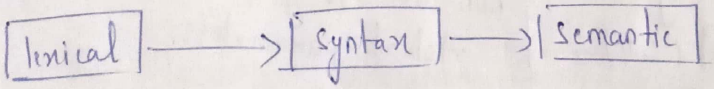


Part - 2 :-

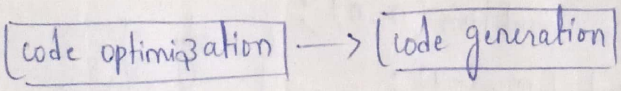
Intermediate code Generator :-



Front end :



Back end :-

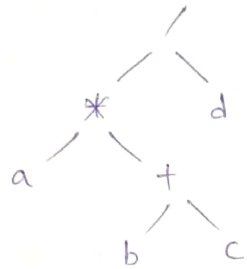


* Forms of ICG :-

1) Syntax tree
2) postfix notation / polish notation
3) 3-addres code
   - Quadraples
   - Triples
   - Indirect triples

1) syntax tree :- Each internal node or parent node represents an operator and each leaf node represents operands like constants.

Ex:- a * (b+c)/d



2) postfix notation :- operator existing after operands.

* operator is a symbol which is used to represent the relation b/w 2 operands.

Ex:- (a+b) * c — ab+c*

a+(b*c) — abc*+

(a-b) * (c/d) — ab-cd/*

3) 3-addres code :-

* Each instruction must contain atmost 3-addreses and atmost 1 operator in R.H.S.

Ex: x+y*z - not a 3-addres code
coz there are more than 2 operators.

∴ $t_1 = y*z$

$t_2 = x+t_1$

(*) quadraples means it contains 4 fields. They are :-

1) operator
2) Argument 1 (left-side operand)
3) Argument 2 (Right side operand)
4) Result

Ex:- $a = b * -c + b * -c$

unary +, - operand will have highest priority

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$a = t_5$

$t_5 = t_2 + t_4$

Address	op	Arg 1	Arg 2	Result
(0)	-	c		t_1
(1)	*	b	t_1	t_2
(2)	-	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5

Here we are wasting memory by using more no. of temporary variables. So we use triples.

* Triples contains only 3 fields :- 1) operator
2) Arg 1
3) Arg 2

result	op	Arg 1	Arg 2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(2)

* Indirect triples contains an extra field known as "pointer field" it points the triples.

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)

pointer field points to triples

100 - (0)
101 - (1)
102 - (2)
103 - (3)
104 - (4)

* Triples and Indirect triples doesn't contain temporary variables.

* Three - Address Instruction forms : (or) Types of 3-address codes ;-

1) Assignment Instruction in the form of $x = y \text{ op } z$
   (op: binary operator)

   Ex:- $x = y+z$
   $x = y>z$
   $x = y \&\& z$
   $x = y \| z$

2) Assignment Instruction in the form of $x = \text{op } y$
   (op: unary operator.)

   Ex:- $x = -y$
   $x = +y$
   $x = !y$

3) Copy Instructions in the form of $x = y$
   where the value of y is assigned to x.

   Ex:- $x = y$
   $y = 10$;
   $x = y$;

4) unconditional jump goto L ;

   Ex:- goto 1000 ;

5) conditional jumps of the form x relop y
   (relop: relational operator.)

   Ex:- if $x>y$ goto L , $x = 100$, $y = 10$

   Ex:- if x goto L_1
   else goto L_2 $x = 1$, $x = 0$

6) procedure calls and return values are implemented.

   Ex:- param x => actual / formal parameter.
   $y = P.n$ → parameter
   return y

7) Address and pointer assignments of the form.

   $x = \&y$
   $x = *y$
   $*x = y$

* Translation of Boolean Expressions or Generating 3-address code for Boolean expressions :

B, B_1, B_2 are Boolean expressions

* In logical OR operator, if B_1, B_2 are false, the result is false. If any one of the Boolean expression is true, i.e B_1 or B_2, the result is true

* Here newlabel is a function, which generates 3-address code for B_1.false

Ex:- $B \rightarrow B_1 \,||\, B_2$

Semantic actions:
- $B_1.true = B.true$
- $B_1.false = newlabel()$
- $B_2.true = B.true$
- $B_2.false = B.false$

$||$ → concatenation symbol

3-address code :- $B_1.code \,||\, label(B_1.false) \,||\, B_2.code$

Ex:- $B \rightarrow B_1 \,\&\&\, B_2$

$B_1.false = B.false$

$B_1.true = newlabel()$ => creates 3-address code

$B_2.true = B.true$

$B_2.false = B.false$

3-address code :- $B.code = B_1.code \,||\, label(B_1.true) \,||\, B_2.code$

Ex:- $B \rightarrow !B_1$

$B_1.true = B.false$

$B_1.false = B.true$

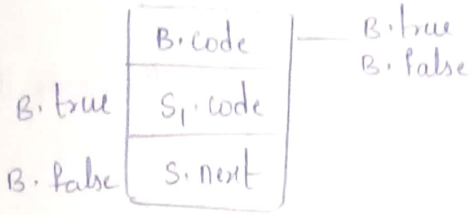
3-address code :- $B.code = B_1.code$

* Intermediate code for flow of control statements :-

1) simple if :- $S \rightarrow$ if (B) then S_1

condition S, S_1 – statements.

code for simple if

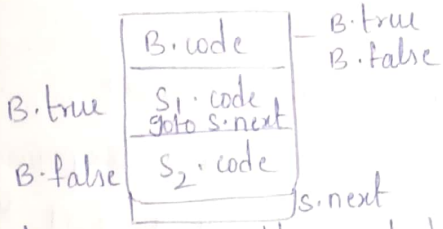


Semantic actions / Rules (3-address code)

B.true = newlabel()
S1.next = S.next
B.false = S.next

* S.code = B.code || label (B.true) || S_1.code.

2) if-else :- $S \rightarrow$ if (B) then S_1 else S_2



Semantic rules (3-address code)

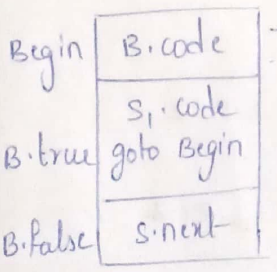
B.true = newlabel()
S_1.next = S.next
B.false = newlabel()
S_2.next = S.next

* S.code = B.code || label (B.true) || S_1.code || gen (goto (S.next)) ||
label (B.false) || S_2.code

generates goto func^n

Both true & false uses the same goto function.

3) While :- $S \rightarrow$ while (B) then S_1



Semantic Rules (3-address code)

Begin = newlabel()
B.true = newlabel()
S_1.next = begin
B.false = S.next

* S.code = label(Begin) || B.code || label(B.true) || S_1.code || gen ('goto' Begin)

* Syntax directed translation to produce 3-address code :-

Grammer	Semantic Actions
$S \rightarrow id = E$	{ gen (id.place = E.place); }
$E \rightarrow E_1 + E_2$	{ E.place = newtemp; gen (E.place = E_1.place + E_2.pla
$E \rightarrow E * E_2$	{ E.place = newtemp; gen (E.place = E_1.place * t_2.place);
$E \rightarrow id$	{ E.place = id.place; }

* Gen :- It'll generate 3-address code in output.
* newtemp :- It is a function used to create a temporary variabl
* place :- It is an attribute which is used to hold the value of the variable.

Ex:- $a = x + y * z$

$t_1 = y * z$
$t_2 = x + t_1$
$a = t_2$

} 3-address code.

SDT :-