Compiler Design
Assignment 1

Q.1 Explain YACC
Ans.
Yacc (Yet Another Compiler-Compiler) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement.

These are some points about YACC:

Input: A CFG- file.y

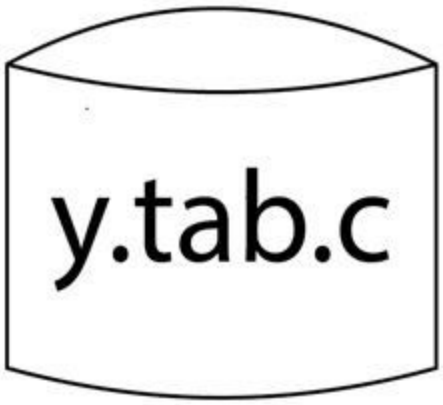
Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

The basic operational sequence is as follows:

This file contains the desired grammar in YACC format.

It shows the YACC program.



It is the c source program created by YACC.

C Compiler

Executable file that will parse grammar given in gram.Y

Q.2 Translation scheme for generation of three address code for Boolean expression
Ans.

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

E → E OR E
E → E AND E
E → NOT E
E → (E)
E → id relop id
E → TRUE
E → FALSE

The relop is denoted by <, >, <, >.

The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Production rule	Semantic actions
E → E1 OR E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3); EMIT (E.place ':=' '0') EMIT ('goto' nextstat + 2) EMIT (E.place ':=' '1') }
E → TRUE	{E.place := newtemp(); Emit (E.place ':=' '1') }
E → FALSE	{E.place := newtemp(); Emit (E.place ':=' '0') }

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three address code using the above translation scheme:

p>q AND r<s OR u>r

100: **if** p>q **goto** 103
101: t1:=0
102: **goto** 104
103: t1:=1
104: **if** r>s **goto** 107
105: t2:=0

106: **goto** 108
107: t2:=1
108: **if** u>v **goto** 111
109: t3:=0
110: **goto** 112
111: t3:= 1
112: t4:= t1 AND t2
113: t5:= t4 OR t3

Q.3 Backpatching
Ans.

Backpatching comes into play in the intermediate code generation step of the compiler. There are times when the compiler has to execute a jump instruction but it doesn't know where to yet. So it will fill in some kind of filler or blank value at this point and remember that this happened. Then once the value is found that will actually be used the compiler will go back "backpatch" and fill in the correct value.

In the process of code generation(mainly, 3 Address Code), all the labels may not be known in a single pass hence, we use a technique called Backpatching.

Backpatching is the activity of filling up *unspecified information of labels* using appropriate semantic actions *during the process of code generation*.

Backpatching Algorithms perform three types of operations

1) makelist (i) – creates a new list containing only i, an index into the array of quadruples and returns a pointer to the list it has made.

2) Merge (i, j) – concatenates the lists pointed to by i and j, and returns a pointer to the concatenated list.

3) Backpatch (p, i) – inserts i as the target label for each of the statements on the list pointed to by p.

here 'p' is the pointer to any statement say p=> "if a<b goto ____".

Initially, we don't know the label to fill just after goto.

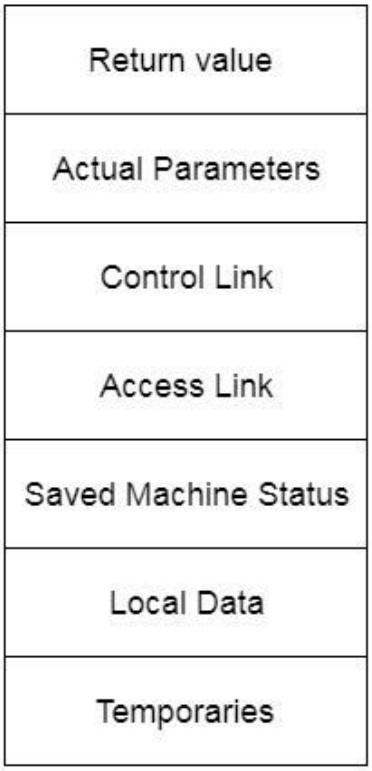
After backpatch(p,i)

p=>"if a<b goto i"

Q.4 What is activation records and storage organisation.
Ans.

Activation Records :
Activation record is used to manage the information needed by a single execution of a procedure. An activation record is pushed into the stack when a procedure is called and it is popped when the control returns to the caller function.

The diagram below shows the contents of activation records:



Return Value**:** It is used by calling procedure to return a value to calling procedure.

Actual Parameter**:** It is used by calling procedures to supply parameters to the called procedures.

Control Link**:** It points to activation record of the caller.

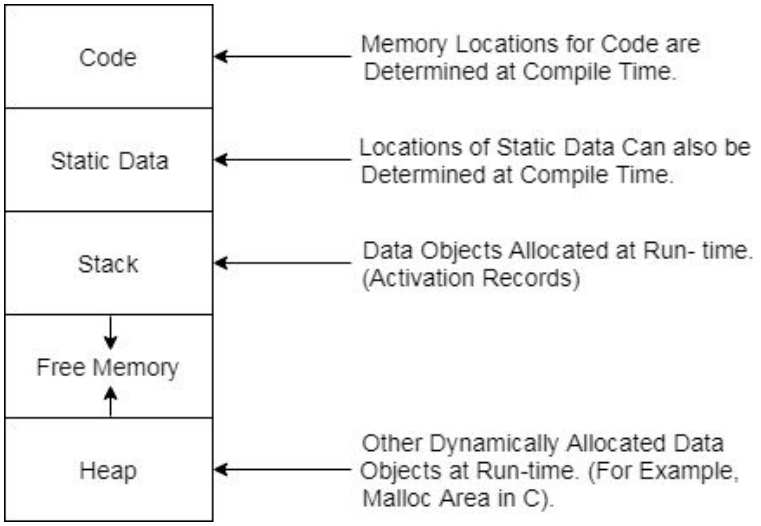
Access Link**:** It is used to refer to non-local data held in other activation records.

Saved Machine Status**:** It holds the information about status of machine before the procedure is called.

Local Data**:** It holds the data that is local to the execution of the procedure.

Temporaries**:** It stores the value that arises in the evaluation of an expression.

Storage Organisation :
When the target program executes then it runs in its own logical address space in which the value of each program has a location. The logical address space is shared among the compiler, operating system and target machine for management and organization. The operating system is used to map the logical address into physical address which is usually spread throughout the memory.



- Runtime storage comes into blocks, where a byte is used to show the smallest unit of addressable memory. Using the four bytes a machine word can form. Object of multibyte is stored in consecutive bytes and gives the first byte address.
- Run-time storage can be subdivide to hold the different components of an executing program:

1. Generated executable code
2. Static data objects
3. Dynamic data-object- heap
4. Automatic data objects- stack

Q.5 Explain storage allocation strategies
Ans.

STORAGE ALLOCATION STRATEGIES
The different storage allocation strategies are :
1. Static allocation - lays out storage for all data objects at compile time
2. Stack allocation - manages the run-time storage as a stack.
3. Heap allocation - allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package. Since the bindings do not change at run-time, every time a procedure is activated, its names are bound to the same storage locations. Therefore values of local names are retained across activations of a procedure.

That is when control returns to a procedure the values of the locals are the same as they were when control left the last time. From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

STACK ALLOCATION OF SPACE

All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack. Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields. A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call. The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).

When designing calling sequences and the layout of activation records, the following principles are helpful:

- Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.
- Fixed length items are generally placed in the middle. Such as the control link, the accesslink, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation
- record. The most common example is dynamically sized array, where the value of one of
- the callee's parameters determines the length of the array.

We must locate the top-of-stack pointer judiciously. A common approach is to have it pointto the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

- The calling sequence and its division between caller and callee are as follows:
- The caller evaluates the actual parameters.
- The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments the top_sp to the respective positions.
- The callee saves the register values and other status information.
- The callee initializes its local data and begins execution.

A suitable, corresponding return sequence is:

- The callee places the return value next to the parameters.
- Using the information in the machine-status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.
- Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller therefore may use that value.

Variable length data on stack:

The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack. The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space. The same scheme works for objects of

any type if they are local to the procedure called and have a size that depends on the parameters of the call.

HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

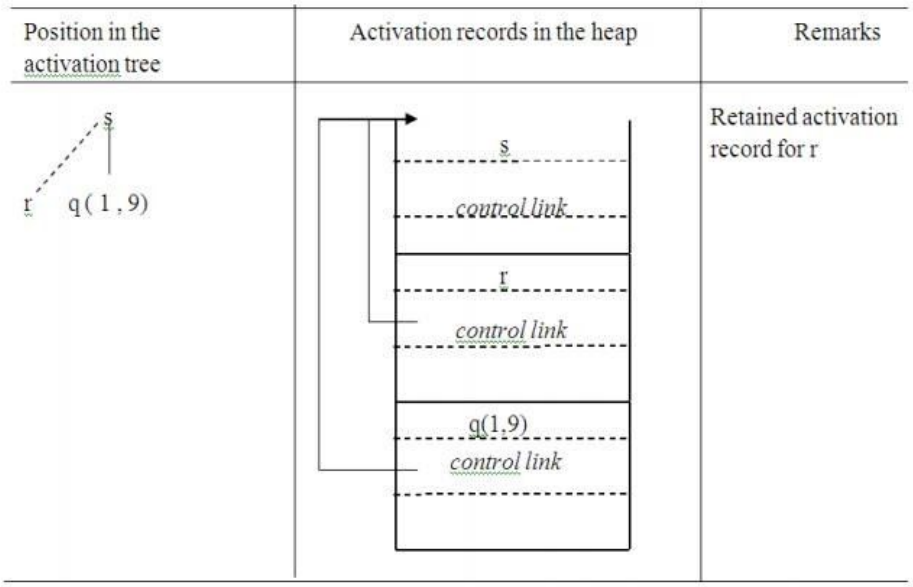


Fig. 2.10 Records for live activations need not be adjacent in heap

Fig. Records for live activations need not be adjacent in heap

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.