Compiler Design

Assignment-2

SUBMITTED TO-

Dr. Aruna Mailk
CSE Department

SUBMITTED BY-

Name – Ankit Goyal
Roll no - 17103011
Group - G-1
Branch - CSE

Q1. For the given code, generate three address code.

```
if(a<b+c*20)
{
        a = a*b-50;
        d = (a/b)+25;
        print(a,d)
}
```

Ans:

```
t1 = a
t2 = b+c
t3 = t2*20
if t1<t3
        t4 = a*b
        t5 – t4-50
        a = t5
        t6 = a/b
        t7 = t6+25
        d = t5
        print(a,d)
```

Q2. Check whether the given grammar is LL(1) or not ?

G: S→Aa|bAc|Bc|bBa, A→ d, B→ d

No left recursion, hence no modification required

First and Follow sets

Non terminals	First	Follow
S	{ b, d }	{$}
A	{d}	{a,c}
B	{d}	{c,a}

Parsing table: String – bdc$

	a	b	c	d	$
S		S → bAc			
A					A → d
B					

Q3. List out and explain the rules to construct simple precedence relation for a context free grammar.

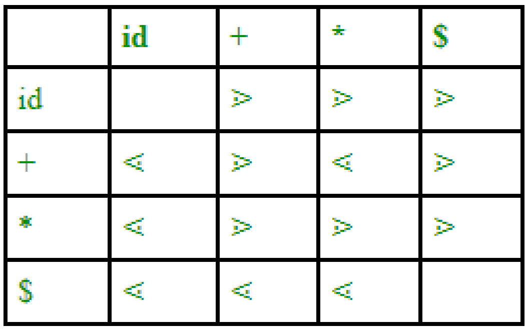
An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in any parser except operator precedence parser.

There are two methods for determining what precedence relations should hold between a pair of terminals:

1. Use the conventional associativity and precedence of operator.
2. The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

This parser relies on the following three precedence relations: **$\lessdot$, $\doteq$, $\gtrdot$**

- **a $\lessdot$ b** This means a "yields precedence to" b.
- **a $\gtrdot$ b** This means a "takes precedence over" b.
- **a $\doteq$ b** This means a "has same precedence as" b



	id	+	*	$
id		$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Q4. Explain different modules used for language processing.

- Preprocessor

  A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

- Interpreter

  An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. Interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it.

- Compiler

A compiler that takes the source code of one programming language and translates it into the source code of another programming language. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. Compiler reads the whole program even if it encounters several errors.

- Assembler
  An assembler translates assembly language programs into machine code.The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

- Linker
  Linker is a computer program that links and merges various object files together in order to make an executable file.

- Loader
  Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it.

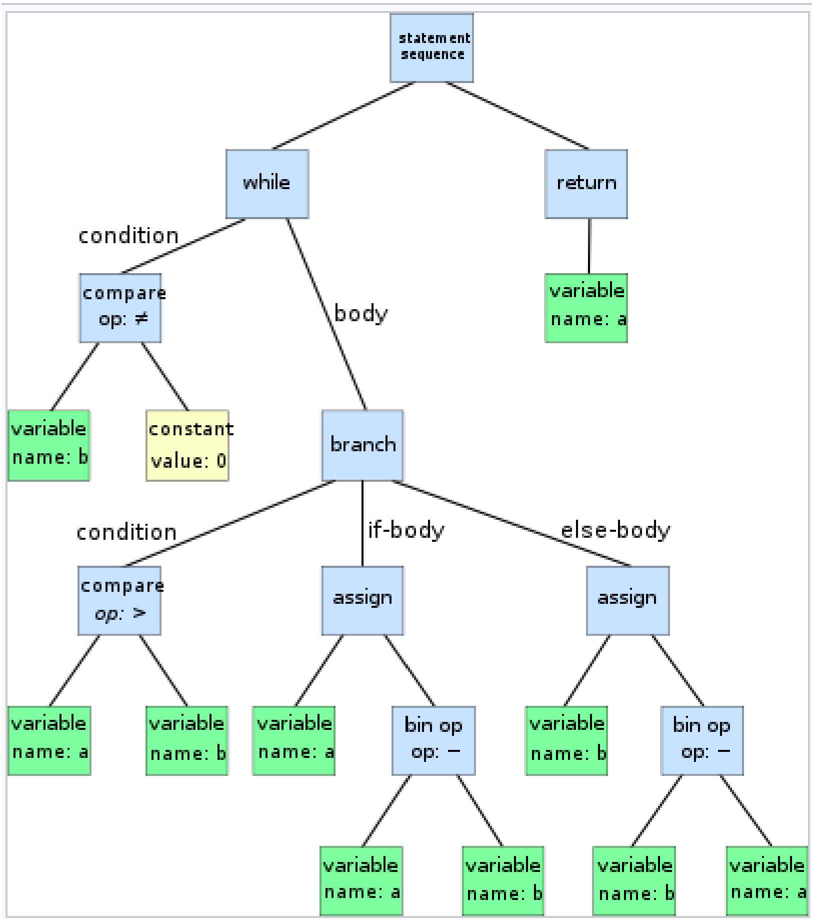
Q5. What is an Abstract syntax tree? How to construct it using mknode(), mkleaf() functions ? Give an example.

Abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

- mknode() : It consists of operators or non-terminals
- mkleaf() : It consists of terminals

Example :

```
while b ≠ 0
       if a > b
         a := a − b
       else
         b := b − a
return a
```


statement sequence
while
return
condition
body
compare
op: ≠
variable
name: b
constant
value: 0
variable
name: a
branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: −
variable
name: b
bin op
op: −
variable
name: a
variable
name: b
variable
name: b
variable
name: a