राष्ट्रीय प्रौद्योगिकी संस्थान, रायपुर (छ.ग.)

National Institute of Technology, Raipur (C.G,)

Digital End Semester Examination(Theory/Practical) Autumn 2021-22

CBCS SCHEME/NIT SCHEME

(Tick Appropriate Scheme)

Roll No.: 19115080

Roll No. (In Words): one nine one one five zero eight zero

Branch: Computer Science & Engineering

Enrolment No.: 190563

Subject: Compiler Design

Subject Code: CS105101CS

Date of Examination: 10/11/2021

Course: B.Tech **Semester:** 5th

Total No. of Pages in This Copy: 9

Student's Name: Abburi Shivaani

Signature of Student:

UNIT - III

Q1. An activation record is a private memory block related to the calling of a procedure. We use them to manage procedure calls, it is a runtime structure.

When the control stack is called, activation starts following which procedure name will be pushed onto the stack and when it returns, the activation ends and will be popped. It is mainly used to control the information required by a single execution.

When a procedure is called the activation record is pushed onto the stack and when the control returns to the caller function, it is popped.

An activation record consists of:

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

Actual Parameter: This field's usage is done by calling procedures to supply parameters to the called procedures.

Control Link: This link points to the activation record of the caller.

Access Link: Its used to connect with non local data stored in other activation records.

Saved Machine Status: Saves the status of the machine before calling of the procedure.

Local Data: Stores data which is local to the execution of the procedure.

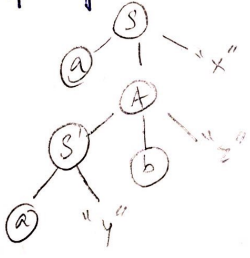
Temporaries: To save the value resulted by the evaluation of the expression.

Return Value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries.

Q2. Given,

$S \rightarrow aA$ { print X }
$S \rightarrow a$ { print Y }
$A \rightarrow Sb$ { print Z }

ip input: aaaaab



Given string: aaaaab

aaaaSb	Z
aaaaA	Y
aaaS	X
SaaS	X
SSaS	X
SSSS	X

Output printed will be "ZYXXXX".

UNIT - II

Q1. Given,

$$X \longrightarrow (Y) \mid t$$
$$Y \longrightarrow Y + X \mid Y * X \mid X$$

String = (t + (t * t))

Below are mentioned are steps to make the operator precedence Parser:

Step 1: Check if given grammar is in operator precedence form, as there is no ε in the right hand side of any production and neither are there any 2 non terminals that are adjacent, this grammar is in operator precedence form.

Step -2: The grammar consists of the following terminal symbols - { (,), t, +, *, $ }.

Therefore, the operator precedence table will be constructed as follows:

	(	)	t	+	*	$
(	>	>	<	>	>	>
)	>	>	<	>	>	>
t	>	>	<	>	>	>
+	<	<	<	>	<	>
*	<	<	<	>	>	>
$	<	<	<	<	<	

Given string: (t + (t * t))

Step 3: On both ends of the string insert '$' symbol making the string, $(t + (t * t)) $

Step 4: Between the string symbols, insert the precedence operators.
Making the string:

$ < (< t > + < (< t > * < t >) >) > $

Step 5: We parse and scan the string:

$ < (< t > + < (< t > * < t >) >) > $
$ < (X + < (< t > * < t >) >) > $
$ < (X + < (X * < t >) >) > $
$ < (X + < (X * X) >) > $
$ < (X + < (Y * X) >) > $
$ < (X + Y) > $
$ < (Y + X) > $
$ < (Y) > $
$ < X > $
$ $

The parsing of string is completed.

UNIT - I

Q 2. (a) Language Preprocessor:

A preprocessor is how a system software designed to perform preprocessing on high level languages. Language processing is the first step of the language process system which is used to translate high level language to machine level language.

The language preprocessor contains all the ~~header~~ header files and analyzes if a macro (code that is assigned a name). Every time the name of the code is used, the prepocessor is replaced by the contents of the macros. using an interpreter or compiler.

(b) Pretty printer: as the name suggests, pretty printer is a code beautifier, it is the process of presenting code in a attractive and organized fashion. It makes the code more comprehensible by adding appropiate line breaks and indentations. The source code's layout is standardized using a pretty printer.

(c) Text Formatter: Compilation consists of 2 parts analysis and synthesis. Basic Text formatters need input strings and indicate commands to indicate paragraphs, and figures. The analysis part of a compiler breaks down the soure londe into smaller pieces and creates an immediate representation of the source program. Text formatters are an example of analysis portion of compilers.

Q1.

Sno.	Lexeme	Attribute of Token	Token
1.	int	Keyword	<key, int>
2.	fact	identifier	<id, fact>
3.	(	operation	<op, (>
4.	int	Keyword	<key, int>
5.	x	identifier	<id, x>
6.	)	operator	<op,)>
7.	{	operator	<op, {>
8.	if	keyword	<key, if>
9.	x	identifier	<id, x>
10.	>	operator	<op, >>
11.	1	integer	<int, 1>
12.	)	operator	<op,)>
13.	return	keyword	<key, return>
14.	x	identifier	<id, x>
15.	*	operator	<op, *>
16.	fact	identifier	<id, fact>
17.	(	operator	<op, (>
18.	x	identifier	<id, x>
19.	-	operator	<op, ->
20.	1	integer	<int, 1>
21.	;	special symbol	<sym, ;>
22.	else	keyword	<key, else>
23.	return	keyword	<key, return>
24.	1	integer	<int, 1>
25.	;	special symbol	<sym, ;>
26.	}	operator	<op, }>
27.	void	keyword	<key, void>
28.	main	keyword	<key, main>
29.	(	operator	<op, (>
30.	void	keyword	<key, void>
31.	)	identifier	<id,)>
32.	{	operator	<op, {>

Sno.	Lexeme	Attribute of Token	Token
33.	int	integer keyword	<key, int>
34.	x	identifier	<id, x>
35.	;	special symbol	<sym, ;>
36.	x	identifier	<id, x>
37.	read	keyword	<key, read>
38.	(	operator	<op, (>
39.	)	operator	<op,)>
40.	;	special symbol	<sym, ;>
41.	if	keyword	<key, if>
42.	(	operator	<op, (>
43.	x	identifier	<id, x>
44.	>	operator	<op, >>
45.	0	integer	<int, 0>
46.	)	operator	<op,)>
47.	write	keyword	<key, write>
48.	(	operator	<op, (>
49.	fact	identifier	<id, fact>
50.	(	operator	<op, (>
51.	x	identifier	<id, x>
52.	)	operator	<op,)>
53.	)	operator	<op,)>
54.	;	special symbols	<sym, ;>
55.	(	operator	<op, (>
56.	)	operator	<op,)>
57.	=	operator	<op, =>

Count of tokens generated by scanner = 57.

UNIT-4

Q1. → We can see that in the 1st for loop, there is $x = y + z$, which is being calculated inside the loop again and again even when the value is not being affected.
Hence, to optimize this we can use CODE MOTION:

```
x = y + z;   st = x * n;
for (int w = 0; w < n; w++)
{   a[w] = 6 * w + st; }
```

→ In the 2nd for loop, we can see that the loop is iteration from 0 to 2100 and increasing by 1 in every step.
This can be optimized by decreasing the number of iterations and increasing the step size. This process is known as loop unrolling.
Here, we can compelete 5 assignment with each iteration, and increase the step size by 5.

```
for (int x=0; x < 2100; x+=5)
{       b[x] = 1;
        b[x+1] = 1;
        b[x+2] = 1;
        b[x+3] = 1;
        b[x+4] = 1;
}
```

→ We can apply loop unrolling in the third loop as well so as to optimize it. We can compelete 5 assignments with every iteration and increase step size by 5.

```
for (int y = 1000; y < 2000; y += 5)
{
    a[y] = 0;
    a[y+1] = 0;
    a[y+2] = 0;
    a[y+3] = 0;
    a[y+4] = 0;
}
```

* Peephole Optimization:

Peephole optimizations implies optimization of small code snippets. It can only be applied on intermediate and target codes. However, the code here is in a high level language and therefore peephole optimize wont be possible.

unit 2

Q2.
```
1  sum = 0
2  i = 1
3  k = 0
4  t1 = k * 5
5  k = t1
6  print k
7  t2 = k + 1
8  k = t2
9  if k < 10 goto 4
10 t3 = 2 * i
11 t4 = a[t3]
12 t5 = sum + t4
13 sum = t5
14 t6 = i + 1
15 i = t6
16 if i <= 10 goto 3
17 avg = sum/i
```