Compiler Design- Exercises

07.11.2025

Q1. Answer True/False – 1 Minute

Both static allocation for data, and stack allocation support recursion.

Q2. Answer True/False- 1 Minute

In heap allocation, memory is allocated at compile-time.

Q3. Choose the correct option- 1 Minute

Which of the following is **not represented** in a function’s activation record frame for a stack-based programming language?

(a) Values of local variable

(b) Return address

(c) Information required to access non-local variables (SL/DL)

(d) Heap area

Q4. choose the correct option- 1 Minute

Heap area is needed for languages that

(a) Support recursion
(b) Support dynamic data structures
(c) None of the above

Q5. choose the correct option- 1 Minute

Which of the following, memory requirements are usually known at compile-time?

(a) Code

(b) Data

(c) None of the above

Q6. choose the correct option- 1 Minute

In which mechanism the calling procedure passes the value of the actual parameter and the compiler puts it in to the called procedure's activation record?

(a) Call by reference
(b) Call by name
(c) Call by value
(d) None of the above

Q7- 4 Minutes

Describe the basic structure of an Activation Record (AR).

Explain briefly (why/why not) if the activation records (ARs) for each of the functions can be allocated **statically** for the following C program.

```
#include <stdio.h>
int ctr = 0;
int func1(int a, int b) {
  int c;
  ctr++;
  if (b==3) return(a*a*a);
  else {
    c = func1(a, b/3);
    return (c*c*c);
  }
}
int main (){
   func1(4, 81);
   printf ("%d", ctr);
}
```

Q8- 6 Minutes

Consider the following code snippet. What will be the value printed when

1. **call-by-value** mechanism is considered
2. **call-by-reference** mechanism is considered
3. **call-by-value-result** mechanism is considered

```
int n;

void p(int k){
   n = n + 1;
   k = k+4;
   print(n);
}

main( ){
  n = 0;
  p(n);
  print(n);
}
```

Q9- 6 Minutes

Consider the following code snippet. What will be the value printed when:

1. *call-by-value* mechanism is considered.
2. *call-by-reference* mechanism is considered.
3. *call-by-name* mechanism is considered.
4. *call-by-value-result* mechanism is considered.

```
int a=10;
void func1(int x){
   a = 100;
   a = a+20;
   x = x+10;
}

int main(){
   func1(a);
   printf("%d\n", a);
}
```

Q10- 6 Minutes

We discussed about ***3-address code*** generation for some high-level programming constructs.
Based on the concepts discussed, **provide 3-address code** corresponding to the following code snippet.

(**NOTE**: it is not necessary to define grammar and SDT scheme etc.
You can directly give the 3-address code corresponding to the given code snippet):

```
a=3;
b=4;
for(i=0;i<n;i++){
        a=b+1;
        a=a*a;
}
c=a;
```

Q11- 6 Minutes

Consider the following Syntax Directed Translation Scheme with non-terminals **{S, W}**, start symbol **S**, and terminals **{x, y, z}**.
For the input "**xxxxyzz**", what will the output printed by a shift-reduce parser using the above SDT scheme? Explain/justify.

$S \rightarrow xxW$	**{print (1);}**
$S \rightarrow y$	**{print (2);}**
$W \rightarrow Sz$	**{print (3);}**

Q12- 5 Minutes

Consider the syntax directed definition shown below.

S → id : = E {gen (id.place = E.place;);}
E → E1 + E2   {t = newtemp ( ); gen (t = El.place + E2.place;); E.place = t}
E → id   {E.place = id.place;}

Here, gen is a function that generates the output code, and newtemp is a function that returns the name of a new temporary variable on every call. Assume that ti's are the temporary variable names generated by newtemp. For the statement 'X: = Y + Z', the 3-address code sequence generated by this definition is

Q13- 5 Minutes

Consider the intermediate code given below.
Identify all basic blocks and draw the control-flow graph.

```
(1) i = 1
(2) j = 1
(3) t1 = 5 * i
(4) t2 = t1+ j
(5) t3 = 4 * t2
(6) t4 = t3
(7) a[t4] = – 1
(8) j = j + 1
(9) if j < = 5 goto (3)
(10) i = i + 1
(11) if i < 5 goto (2)
```

Q14- 3 Minutes

Let us consider the following fragment of attribute grammar.
(“**sy**” is a ***synthesized*** attribute, and “**in**” is an ***inherited*** attribute).

$$
\begin{array}{lll}
S \rightarrow A\ B & \{\ A.in = B.sy; & S.sy = A.sy\ \} \\
A \rightarrow a & \{\ A.sy = 1\ \} & \\
B \rightarrow b & \{\ B.sy = 2\ \} &
\end{array}
$$

Is the above grammar **S-attributed** (justify why/why not)? Is the above grammar **L-attributed** (justify why/why not)?

Q15- 3 Minutes

Let us consider the following fragment of attribute grammar.

("**sy**" is a ***synthesized*** attribute, and "**in**" is an ***inherited*** attribute).

$S \rightarrow A\ B$	{ A.in = S.in; B.in = A.in + 1; S.sy = B.in }
$A \rightarrow a$	{ A.sy = 1 }
$B \rightarrow b$	{ B.sy = 1 }

Is the above grammar **S-attributed** (justify why/why not)? Is the above grammar **L-attributed** (justify why/why not)?