**International Institute of Professional Studies,**

**Devi Ahilya Vishwavidyalaya, Indore**

**MCA (6 Years) IX Semester**

**Session July-Dec, 2020**

**IC-901A Compiler Design**

**List of Assignments**

**Date of Submission: 10th February, 2021.**

**Mode of Submission: Online (Link will remain open till 5 pm on 10th February, 2021)**

**Note: Write the answers, scan the sheets and upload in pdf format. Make all entries like Roll Number, Name, Class, Subject Name, Date of Submission and signature on the assignment sheet.**

**Name of the pdf file should be your roll number followed by first name (e. g. 84\_Vijay)**

1. Find the host language and target language for the compilers of the programming languages – FORTRAN, C, C++ and Java.

Ans:- **Java Language compiler**

Source Language: Java Target Language: Bytecode Host Language: C+

1. Explain the concept of Runtime Environment.
2. Identify the lexemes that make up the tokens in the following program:

int max ( i, j ) int i, j;

/\* Return maximum of integers i and j \*/

{

return i > j ? i : j;

}

1. Consider the following grammar

E → E **or** T | T

T → T **and** F | F

F → **not** F | (E) | **true** | **false**

* 1. Construct a parse tree for the sentence **not ( true or flase)**
  2. Show that this grammar generates all Boolean expressions.
  3. Is this grammar ambiguous? Why?

1. Consider the following grammar:

S → aSbS | bSaS | ∈

Construct a recursive - descent parser with backtracking for the grammar. Can a predictive parser be generated for this grammar?

1. Construct an SLR parsing table for the following grammar:

E → E **or** T | T

T → T **and** F | F

F → **not** F | (E) | **true** | **false**

1. Consider the following grammar:

**E → E + T | T**

**T → T F | F**

**F → F\* | a | b**

Construct the **SLR** parsing table for this grammar.

1. Show that no LR (1) grammar is ambiguous.
2. Show that the following grammar is LR (1) but not LALR (1):

**S → Ac | bAc | Bc | bBa**

**A → d**

**B → d**

1. For the input expression **(4\* 7+1) \* 2**, construct an annotated parse tree according to the following syntax-directed definition

|  |  |
| --- | --- |
| Production | Semantic Rules |
| L → E n | *print* (E.*va*l) |
| E → E1 + T | E.*val* := E1.*val* + T.*val* |
| E → T | E.*val* := T.*val* |
| T → T1 \* F | T.*val* := T1.*val* \* F.*val* |
| T → F | T.*val* := F.*val* |
| F → ( E ) | F.*val* := E.*val* |
| F → **digit** | F.*val* := **digit** .*lexval* |

1. Construct the parse tree and syntax tree for the expression ((a) + (b)) according to
   1. the following syntax-directed definition:

|  |  |
| --- | --- |
| Production | Semantic Rules |
| E → E1 + T | E.*nptr* := *mknode* (‘+’, E1.*nptr*, T.*nptr*) |
| E → E1 - T | E.*nptr* := *mknode* (‘-’, E1.*nptr*, T.*nptr*) |
| E → T | E.*nptr* :=T. *nptr* |
| T → ( E ) | T.*nptr* := E.*nptr* |
| T → **id** | T.*nptr* := *makeleaf* (**id, id.***entry*) |
| T → **num** | T.*nptr* := *makeleaf* (**num, num.***val*) |

* 1. the following translation scheme:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| E | **→** | T |  | { R.*i* := T.*nptr* } |
|  |  | R |  | { E. *nptr* := R.*s* } |
| R | → | + |  |  |
|  |  | T |  | { R1.*i* := *mknode* (‘+’, R.*i*, T.*nptr*) } |
|  |  | R1 |  | { R.*s* := R1.*s* } |
| R | → | - |  |  |
|  |  | T |  | { R1.*i* := *mknode* (‘-’, R.*i*, T.*nptr*) } |
|  |  | R1 |  | { R.*s* := R1.*s* } |
| R | → | ∈ |  | { R.*s* := R.*i* } |
| T | → | ( |  |  |
|  |  | E |  |  |
|  |  | ) |  | T. *nptr* := E. *nptr* |
| T | → | **id** |  | {T.*nptr* := *makeleaf* (**id, id.***entry*) } |
| **T** | → | **num** |  | T.*nptr* := *makeleaf* (**num, num.***val*) |

1. Translate the arithmetic expression **a\*-(b+c)** into
   1. a syntax tree
   2. postfix notation
   3. three-address code
2. Translate the expression **–(a+b)\*(c+d) + (a+b+c)** into
   1. quadruples
   2. triples
   3. indirect triples
3. Translate the executable statements of the following C program

**main()**

**{**

**int i;**

**int a[10];**

**i = 1;**

**while ( i <= 10) {**

**a [i] = 0;**

**i = i + 1;**

**}**

**}**

* 1. a syntax tree
  2. postfix notation
  3. three-address code

1. Generate optimal code for the following assignment statements:
   1. **x = a + b \* c**
   2. **x = ( a \* -b) + ( c – ( d + e ) )**
   3. **x = ( a / b – c ) / d**
   4. **x = a + ( b + c / d \* e ) / ( f \* g – h \* i)**
   5. **a [ i, j ] = b [ i, j ] – c [ a [ k, l ] ] \* d [ i + j ]**