LAB REPORT-3
Subtopic: Semantic Analysis

A) Generate syntax directed translations for your grammar.

B) Create a syntax tree for your input program.

C) Show that your compiler is able to detect semantic errors which was not detected up to Syntax Analysis phase.

A) Generate syntax directed translations for your grammar.

```
constant : T_Number {insertRecord("Constant", $<text>1, @1.first_line, currentScope);
           $$ = createID_Const("Constant", $<text>1, currentScope);}
         | T_String {insertRecord("Constant", $<text>1, @1.first_line, currentScope);
           $$ = createID_Const("Constant", $<text>1, currentScope);};
```

The 'constant' non-terminal branches into 'T_Number' and 'T_String' alternatives. Both branches invoke 'insertRecord' and 'createID_Const' functions, handling semantic actions like inserting constant values into a symbol table and creating identifier nodes. '@1.first_line' denotes the line number, and 'currentScope' refers to the symbol table scope. This tree captures parsing and processing of constant declarations.

```
arith_exp : term {$$=$1;}
          | arith_exp  T_PL  arith_exp {$$ = createOp("+", 2, $1, $3);
            $$->val=$1->val+$3->val; }
          | arith_exp  T_MN  arith_exp {$$ = createOp("-", 2, $1, $3);
            $$->val=$1->val-$3->val; }
          | arith_exp  T_ML  arith_exp {$$ = createOp("*", 2, $1, $3);
            $$->val=$1->val*$3->val; }
          | arith_exp  T_DV  arith_exp {$$ = createOp("/", 2, $1, $3);
            $$->val=$1->val/$3->val; }
```

Each rule includes actions embedded within curly braces, denoted by $$, which manipulate attributes associated with nodes. For instance, in the first rule, the term's attribute is assigned to the synthesized attribute of the non-terminal node. Similarly, in subsequent rules, actions create operation nodes with appropriate operators and operands, along with computing and assigning values to these nodes based on the corresponding arithmetic operations.

```
assign_stmt : T_ID T_EQL arith_exp {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);
              $$ = createOp("=", 2, createID_Const("Identifier", $<text>1, currentScope), $3);updateval($<text>1,$$->id->val);}
            | T_ID T_EQL bool_exp {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);
              $$ = createOp("=", 2, createID_Const("Identifier", $<text>1, currentScope), $3);;updateval($<text>1,$$->id->val);}
            | T_ID  T_EQL func_call {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);
              $$ = createOp("=", 2, createID_Const("Identifier", $<text>1, currentScope), $3);;updateval($<text>1,$$->id->val);}
            | T_ID T_EQL T_OB T_CB {insertRecord("ListTypeID", $<text>1, @1.first_line, currentScope);
              $$ = createID_Const("ListTypeID", $<text>1, currentScope);} ;
```

Each production rule defines actions associated with constructing an annotated parse tree. In this specific case, the rule handles assignment statements. The actions include inserting records into a symbol table, creating nodes in the parse tree to represent assignments, and updating associated values in the symbol table. These actions facilitate the process of parsing source code and generating a representation for subsequent semantic analysis.

```
void updateval(char term$<text>1,int val$$->id->val){
    record* n = findRecord(name, scope);
    n->val=val;
}
```

The node representing the "updateval" function would have annotations indicating its return type, parameters, and their types. The "char term$<text>1" parameter suggests a character array named "term" with some additional constraints (perhaps related to its length or format). The "int val$$->id->val" parameter indicates an integer value that is assigned to a member called "val" within a nested structure. The function implementation within the node would reference a record identified by "name" and "scope" parameters, updating its "val" field with the provided value.

Sample Input

x=20
y=5
z=4
print(x*y+z)

Annotated Parse Tree (SDT) Output:

```
StartParse
  |
  |--- assign_stmt
  |       |
  |       |--- T_ID (id='x')
  |       |--- T_EQL
  |       |--- T_NUMBER (Num=20, Value=20)
  |
  |--- assign_stmt
  |       |
  |       |--- T_ID (id='y')
  |       |--- T_EQL
```

```
|      |--- T_NUMBER (Num=5, Value=5)
|
|--- assign_stmt
|      |
|      |--- T_ID (id='z')
|      |--- T_EQL
|      |--- T_NUMBER (Num=4, Value=4)
|
|--- print_stmt
|      |
|      |--- T_Print (Value=104)
|      |--- T_OP
|      |--- arith_exp (Value=104)
|             |
|             |--- arith_exp (Value=100)
|                        |--- arith_exp (Value=20)
|                               |--- term (Value=20)
|                                      |--- T_ID (id='x', Value=20)
|                        |--- T_ML
|                        |--- arith_exp (Value=5)
|                               |--- term (Value=5)
|                                      |--- T_ID (id='y', Value=5)
|             |--- T_PL
|             |
|             |--- arith_exp (Value=4)
|                        |--- term (Value=4)
|                               |--- T_ID (id='z', Value=4)
|      |--- T_CP
```

B) Create a syntax tree for your input program.

For the Syntax Tree, we have 2 Types of Nodes, Leaf nodes and Internal nodes. The nodes can have variable number of children (0-3) depending upon the construct it represents. Take the example of the If-Else Statement,

If
Condition CodeBlock Else

To display the Syntax Tree, we store it as a matrix of levels. We have printed each level of the Syntax Tree. All Internal nodes also have a number enclosed in brackets next to them, which represents the number of children they have in the next level. Leaf nodes in the Syntax Tree representing identifiers, constants, Lists, packages point to a record in the symbol table.

```
typedef struct ASTNode
{
    int nodeNo;
    /*Operator*/
    char *NType;
    int noOps;
    struct ASTNode** NextLevel;
    /*Identifier or Const*/
    record *id;
} node;
```

This ASTNode structure takes care of both leaf nodes as well as Internal “Operator” Nodes. The respective values are set depending upon the type of node. Each node can have 0-3 children. We print the AST by first storing it in a Matrix of Order “MAXLEVELS” x “MAXCHILDREN” and printing the matrix Levelwise. This Matrix is a matrix of pointers to the AST. The “noOps” element of the Node gives the number of children of that node.

```
node *createID_Const(char *type, char *value, int scope)
{
  printf("CIC: %d, %s\n", scope, value);
  node *newNode;
  newNode = (node*)calloc(1, sizeof(node));
  newNode->NType = NULL;
  newNode->noOps = -1;
  newNode->id = findRecord(value, type, scope);
  newNode->nodeNo = nodeCount++;
  prev = root_node;
  root_node = newNode;
  return newNode;
}
```

This function is responsible for creating nodes in a syntax tree to represent identifiers and constants encountered during parsing. It allocates memory for the node, initializes its fields, retrieves information about the identifier or constant from some symbol table or records structure, updates global variables to maintain the syntax tree structure, and returns a pointer to the newly created node.

```
node *createOp(char *oper, int noOps, ...)
{

  printf("CO: %s, %d\n", oper, noOps);
  va_list params;
  node *newNode;
  int i;
  newNode = (node*)calloc(1, sizeof(node));

  newNode->NextLevel = (node**)calloc(noOps, sizeof(node*));

  newNode->NType = (char*)malloc(strlen(oper)+1);
  strcpy(newNode->NType, oper);
  newNode->noOps = noOps;
  va_start(params, noOps);

  for (i = 0; i < noOps; i++)
    newNode->NextLevel[i] = va_arg(params, node*);

  va_end(params);
  newNode->nodeNo = nodeCount++;
  if (start==NULL){
      start=newNode;
  }
  return newNode;
}
```

The function described facilitates the creation of nodes in a syntax tree during parsing, taking three parameters: the operator or operation type, the number of operands or child nodes, and variable pointers to child nodes. Memory is allocated for the new node and its associated data. Arguments are handled to populate the child nodes of the new node. Node numbering and insertion into the tree are managed, with special consideration for the root node. Finally, the function returns a pointer to the newly created node.

```
void ASTToArray(node *root, int level)
{

 if(root == NULL )
 {
   return;
 }
 printf("ATA: %d %d %d %d\n", nodeCount, root->nodeNo, level, root->noOps);
 if(root->noOps <= 0)
 {
   Tree[level][levelIndices[level]] = root;
   levelIndices[level]++;
 }

 if(root->noOps > 0)
 {
      int j;
      Tree[level][levelIndices[level]] = root;

      // printf("ATA: %d %d\n", level, root->noOps);
      levelIndices[level]++;

   for(j=0; j<root->noOps; j++)
   {
      ASTToArray(root->NextLevel[j], level+1);
   }
 }
}
```

This function traverses the AST in a depth-first manner and populates a two-dimensional array Tree with pointers to the nodes of the AST, organizing them based on their levels or depths in the tree.

```
void printAST(node *root)
{
    printf("\n-------------------------Abstract Syntax Tree---------------
    ASTToArray(start, 0);
    printf("Done converting AST to Array\n");
    int j = 0, p, q, maxLevel = 0, lCount = 0;

    while(levelIndices[maxLevel] > 0) maxLevel++;

    while(levelIndices[j] > 0)
    {
        for(q=0; q<lCount; q++)
        {
            printf(" ");

        }
        for(p=0; p<levelIndices[j] ; p++)
        {
            if(Tree[j][p]->noOps == -1)
            {
                printf("%s  ", Tree[j][p]->id->name);
                lCount+=strlen(Tree[j][p]->id->name);
            }
            else if(Tree[j][p]->noOps == 0)
            {
                printf("%s  ", Tree[j][p]->NType);
                lCount+=strlen(Tree[j][p]->NType);
            }
            else
            {
                printf("%s(%d) ", Tree[j][p]->NType, Tree[j][p]->noOps);
            }
        }
        j++;
        printf("\n");
    }
}
```

The process begins with printing a header to signify the subsequent output as the abstract syntax tree (AST). The AST is then converted into an array for easier traversal and organization. Next, the maximum depth of the AST is determined by counting nodes at each level, crucial for understanding its structure. Iterating through each level, the tree structure is printed, adjusting indentation to reflect hierarchy visually. Each node's information, including type and operand count, is then printed, with formatting adjusted based on node type for clarity.

```
arith_exp : term {$$=$1;}
          | arith_exp  T_PL  arith_exp {$$ = createOp("+", 2, $1, $3);}
          | arith_exp  T_MN  arith_exp {$$ = createOp("-", 2, $1, $3);}
          | arith_exp  T_ML  arith_exp {$$ = createOp("*", 2, $1, $3);}
          | arith_exp  T_DV  arith_exp {$$ = createOp("/", 2, $1, $3);}
          | bool_term T_NEQ arith_exp {{$$ = createOp("!=", 2, $1, $3);}}
          | T_MN arith_exp {$$ = createOp("-", 1, $2);}
          | T_OP arith_exp T_CP {$$ = $2;} ;


bool_exp : bool_term T_Or bool_term {$$ = createOp("or", 2, $1, $3);}
         | arith_exp T_LT arith_exp {$$ = createOp("<", 2, $1, $3);}
         | bool_term T_And bool_term {$$ = createOp("and", 2, $1, $3);}
         | bool_term T_NEQ bool_term {$$ = createOp("!=", 2, $1, $3);}
         | arith_exp T_EQ arith_exp {$$ = createOp("==", 2, $1, $3);}
         | arith_exp T_GT arith_exp {$$ = createOp(">", 2, $1, $3);}
         | arith_exp T_ELT arith_exp {$$ = createOp("<=", 2, $1, $3);}
         | arith_exp T_EGT arith_exp {$$ = createOp(">=", 2, $1, $3);}
         | arith_exp T_In T_ID {checkList($<text>3, @3.first_line, currentScope); $$ = createOp("in", 2, $1, createID_Const("Constant", $<text>3, current
         | bool_term {$$=$1;};
```

These yacc rules define the production rules for generating abstract syntax trees (AST) for arithmetic and boolean expressions. The 'arith_exp' rule encompasses arithmetic operations like addition, subtraction, multiplication, division, negation, and parentheses, utilizing the createOp function to generate corresponding AST nodes. Similarly, the 'bool_exp' rule handles boolean operations such as logical OR, logical AND, inequality, equality, comparison operators, and membership tests, also employing the createOp function to construct the respective AST nodes. Ultimately, the resulting AST node for each expression type is assigned to $$.

```
import_stmt : T_Import T_ID {insertRecord("PackageName", $<text>2, @2.first_line, currentScope);  $$ = createOp("import", 1, createID_Const("PackageName"
pass_stmt : T_Pass  {insertRecord("Pass", "Pass", @1.first_line, currentScope);$$ = createOp("pass", 0);};
break_stmt : T_Break {insertRecord("Break", $<text>1, @1.first_line, currentScope);$$ = createOp("break", 0);};
return_stmt : T_Return T_ID {insertRecord("return","return", @1.first_line, currentScope);$$ = createOp("return", 0);};;
            | T_Return constant {insertRecord("return","return", @1.first_line, currentScope);$$ = createOp("return", 0);};;
            | T_Return arith_exp {insertRecord("return","return", @1.first_line, currentScope);$$ = createOp("return", 0);};;
            | T_Return bool_exp {insertRecord("return","return", @1.first_line, currentScope);$$ = createOp("return", 0);};;
            | T_Return func_call {insertRecord("return","return", @1.first_line, currentScope);$$ = createOp("return", 0);};;
            | T_Return T_OB T_CB {insertRecord("return","return", @1.first_line, currentScope);$$ = createOp("return", 0);};;
            | T_Return T_OB constant T_CB {insertRecord("return","return", @1.first_line, currentScope);$$ = createOp("return", 0);};;
            | T_Return {insertRecord("return","return", @1.first_line, currentScope);};;


assign_stmt : T_ID T_EQL arith_exp {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);$$ = createOp("=", 2, createID_Const("Identifier",
            | T_ID T_EQL bool_exp {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);$$ = createOp("=", 2, createID_Const("Identifier", $
            | T_ID  T_EQL func_call {insertRecord("Identifier", $<text>1, @1.first_line, currentScope);$$ = createOp("=", 2, createID_Const("Identifier",
            | T_ID T_EQL list {insertRecord("ListTypeID", $<text>1, @1.first_line, currentScope);$$ = createID_Const("ListTypeID", $<text>1, currentScope
```

The provided yacc rules outline the translation process of various statements and expressions into abstract syntax tree (AST) nodes using the createOp function. Each rule corresponds to a specific statement type, such as import, pass, break, return, and assignment statements. For instance, the import_stmt rule handles import statements by adding a record to the symbol table for the imported package and creating an AST node with the operation "import" and the package name as its operand. The return_stmt rule creates an AST node with the operation "return" and zero or one operand depending on the presence of an expression. Lastly, the assign_stmt rule deals with assignment statements, adding a record for the identifier being assigned and constructing an AST node with the operation "=" and two operands: the identifier and the expression being assigned.

```
if_stmt : T_If bool_exp T_Cln start_suite {insertRecord("If", "If", @1.first_line, currentScope);$$ = createOp("If", 2, $2, $4);}
        | T_If bool_exp T_Cln start_suite elif_stmts {insertRecord("If-Elif", "If-Elif", @1.first_line, currentScope);$$ = createOp("If", 3, $2, $4, $5);

elif_stmts : else_stmt {$$= $1;}
           | T_Elif bool_exp T_Cln start_suite elif_stmts {$$= createOp("Elif", 3, $2, $4, $5);};

else_stmt : T_Else T_Cln start_suite {insertRecord("Else", "Else", @1.first_line, currentScope);$$ = createOp("Else", 1, $3);};

while_stmt : T_While bool_exp T_Cln start_suite {insertRecord("While", "While", @2.first_line, currentScope);$$ = createOp("While", 2, $2, $4);};
```

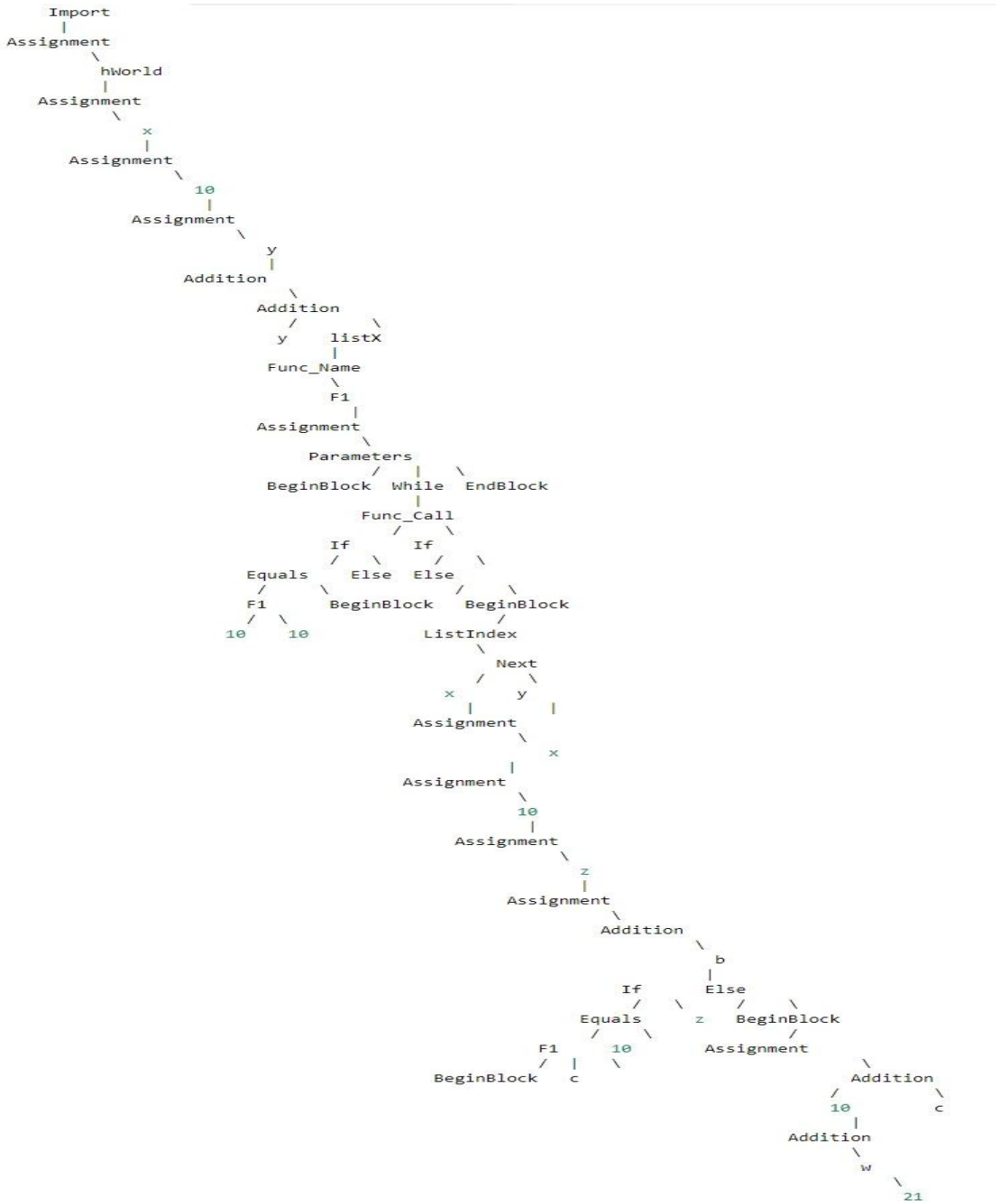
The AST structure for conditional statements in programming languages encompasses if, elif, and else blocks, as well as while loops. An if statement node consists of a boolean expression and a suite of statements to execute if true, potentially followed by elif statements. Each elif block includes a boolean expression and its corresponding suite of statements. The else block represents the suite of statements to execute if none of the preceding conditions are met. Additionally, a while loop node contains a boolean expression for its condition and a suite of statements to execute repeatedly while the condition holds true.

```
call_args : T_ID {addToList($<text>1, 1);} call_list {$$ = createOp(argsList, 0); clearArgsList();}
                  | T_Number {addToList($<text>1, 1);} call_list {$$ = createOp(argsList, 0); clearArgsList();}
                  | T_String {addToList($<text>1, 1);} call_list {$$ = createOp(argsList, 0); clearArgsList();}
                  | {$$ = createOp("Void", 0);};

func_def : T_Def T_ID {insertRecord("Func_Name", $<text>2, @2.first_line, currentScope);} T_OP args T_CP T_Cln start_suite
         {$$ = createOp("Func_Name", 3, createID_Const("Func_Name", $<text>2, currentScope), $5, $8);};

func_call : T_ID T_OP call_args T_CP {$$ = createOp("Func_Call", 2, createID_Const("Func_Name", $<text>1, currentScope), $3);};
```

In the call_args production, AST nodes are generated for identifiers, numbers, or strings followed by additional arguments, forming a hierarchical structure. These nodes represent the arguments passed to functions, facilitating the construction of a complete AST. Similarly, the func_def rule constructs AST nodes for function definitions, capturing essential information such as the function name, arguments, and the suite of statements within the function body. Furthermore, the func_call production rule generates AST nodes representing function calls, incorporating the function name and its arguments, thus enabling the representation of function invocations within the AST.

Syntax Tree Output:

```
import math
x=10
y=20
#Comment1
listX = []
def F1(A, B, C):
        while(listX[2]==y):
                z=10
                b=10-z
                if(z==b):
                        c=10+b
                else:
                        c=20+z
                y=21
```



C) Show that your compiler can detect semantic errors

```
void yyerror(const char *msg)
{
    printf("\nSemantic Error at Line %d, Column : %d\n",  yylineno, yylloc.last_column);
    exit(0);
}
```

The provided function yyerror serves as an error handling mechanism within a compiler or parser. When called, it prints a semantic error message along with the line number and last column where the error occurred, utilizing the printf function.

Sample Error Program

```
import random
x=10
def=5
while(xx<5):
  print(x+12)
```

```
------------------------------Semantic Output--------------------------
T_IMPT T_random
T_NL
2 T_x T_Assign T_10
T_NL
T_Def T_Assign
Semantic Error at Line 3, Column : 4
T_NL
T_While T_OP T_xx T_LT Identifier 'xx' at line 4 Not Declared
T_NL
5 T_Print T_OP T_x T_Plus T_12 T_CP
T_NL
```

Our parser detects two main categories of semantic errors during compilation:

- undeclared variables (line 4) and
- reserved identifier misuse (line 3).

Undeclared variables occur when a variable is referenced without being previously declared within the scope. This error highlights potential issues in program logic and variable scoping. Reserved identifier misuse refers to the improper usage of reserved keywords or identifiers within the programming language, violating language syntax rules. By identifying and flagging these errors, our parser enhances code reliability by producing syntactically and semantically correct programs.