Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Building a Mini Compiler - Abstract Syntax Tree

Preet Kanwal

Department of Computer Science & Engineering

Implementation tasks

- Generating an abstract syntax tree for a single expression.
- Keep track of relationship between various nodes of the tree.

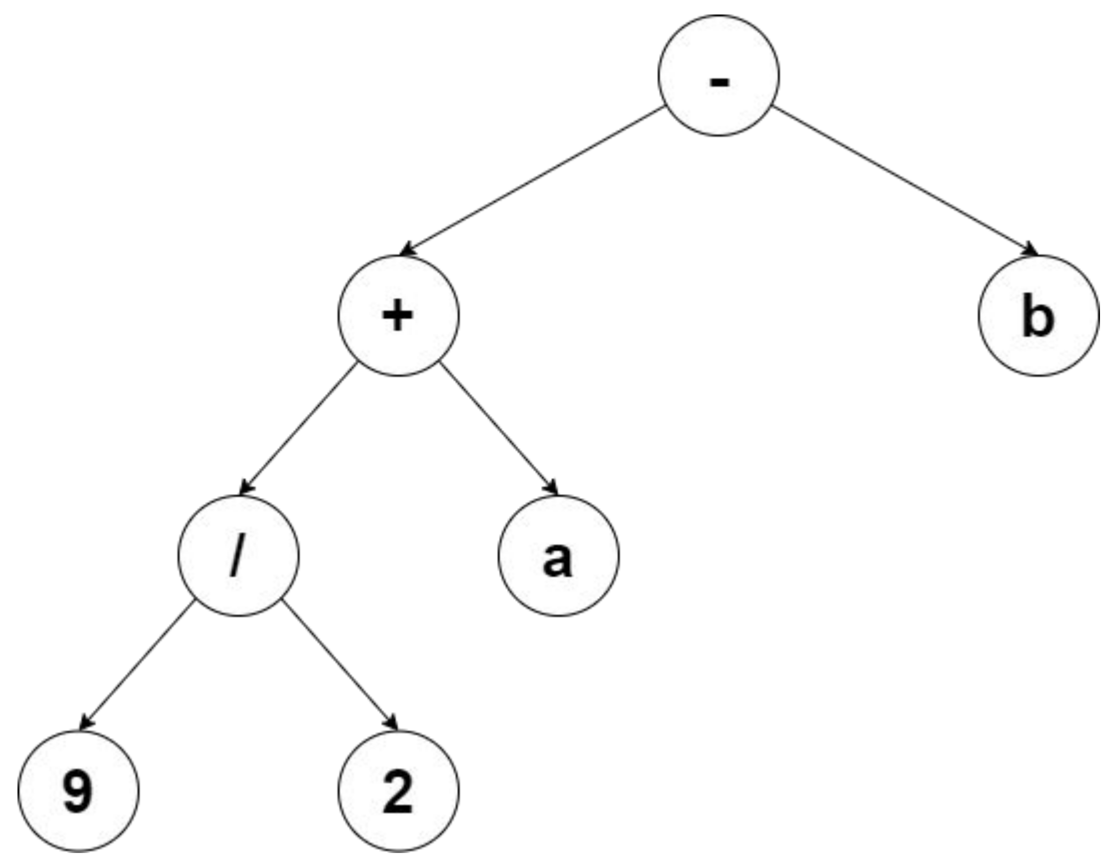
Expected Results

Sample Input:

```
x = 9 / 2 + a - b
```

Output

```
-
+
/
9
2
a
b
```



Note:

- Each line indicates a new node.
- Nodes at printed in preorder

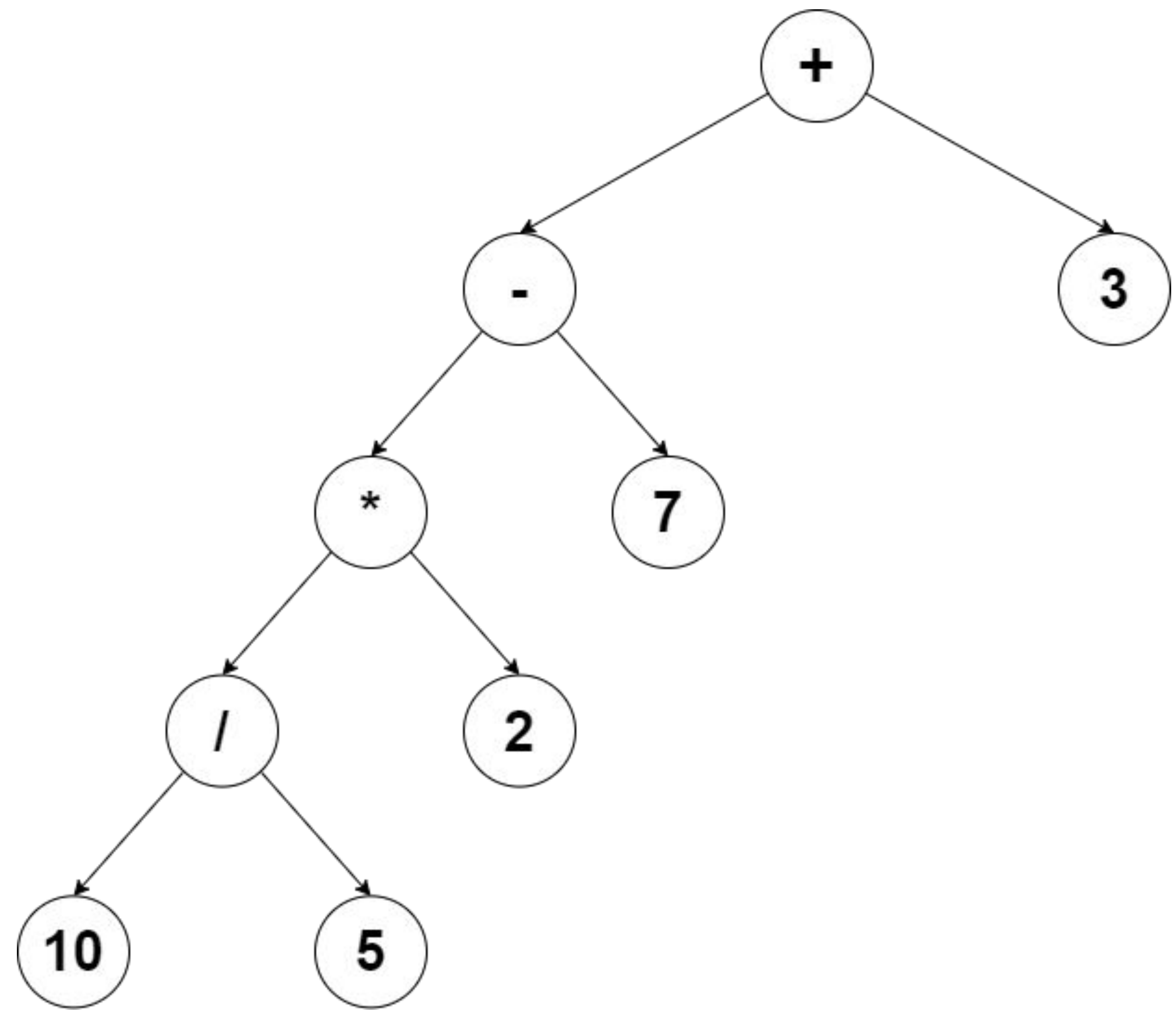
Expected Results

Sample Input:

```
a = 10 / 5 * 2 - 7 + 3
```

Output

```
+
-
*
/
10
5
2
7
3
```



Symbol table implementation: yacc file

- A binary tree can be used to implement the Abstract Syntax Tree.
- An AST is generated only for a valid expression.

```
ASSGN : T_ID '=' E  {
                          // display the created AST
                    }
```

```
E : E '+' T     {
                    /* create a new node for the AST with '+' as the value and
                    assign E as the left and T as the right subtree */
                }
  | E '-' T     {
                    /* create a new node for the AST with '-' as the value and
                    assign E as the left and T as the right subtree */
                }
  | T           {
                    // pass previously created node to parent
                }
  ;

T : T '*' F     {
                    /* create a new node for the AST with '*' as the value and
                    assign T as the left and F as the right subtree */
                }
  | T '/' F     {
                    /* create a new node for the AST with '/' as the value and
                    assign T as the left and F as the right subtree */

                }
  | F           {
                    // pass previously created node to parent
                }
  ;
```

```
F : '(' E ')'    {
                           // pass previously created node to parent
                       }
    | T_ID       {
                       /* create a new node for the AST with T_ID as the value and
                       assign left and right subtree as NULL */
                }
    | T_NUM      {
                       /* create a new node for the AST with T_NUM as the value and
                       assign left and right subtree as NULL */
                   }
    ;
```

Expression Evaluation implementation: yacc file

- Expression grammar can extend to multiple rules, to connect the right nodes, you need to create child nodes and pass them to the parent nodes.

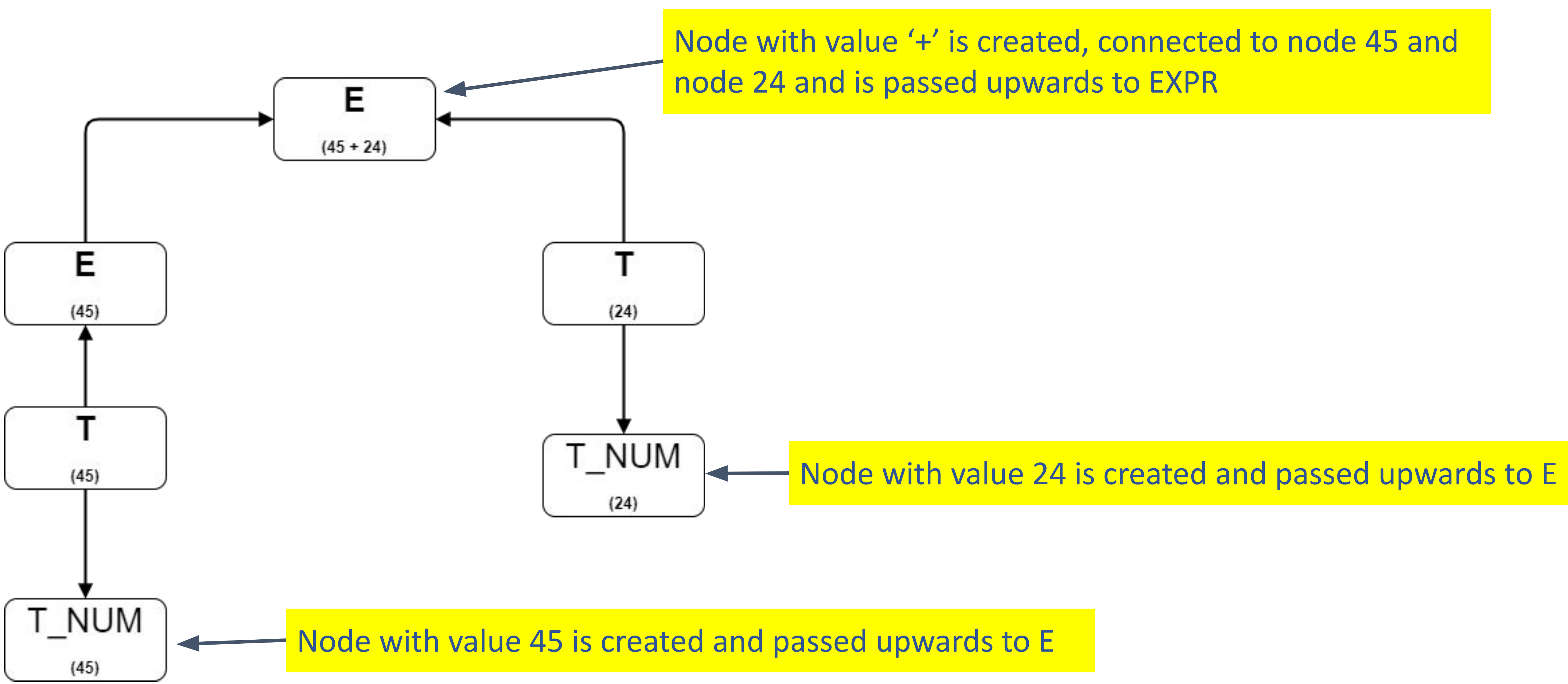
- It is similar to the process used to generate the symbol table. The required values are passed from lower rules to the root.

- Let's take an example expression:
  45 + 24;

- Let the grammar be:
  E: E '+' T | T
  T: T_NUM

Important points:

- You will have to create helper functions for the abstract syntax tree. (example: initialise nodes and assign children)
- You will have to make use of additional structures in the yacc file in order to pass values and addresses to the parent nodes.
- In the previous lab, all grammar rules returned strings, however in this lab you will need to make use of other types (i.e. structures) as well


Node with value ‘+’ is created, connected to node 45 and node 24 and is passed upwards to EXPR
E
(45 + 24)
E
(45)
T
(24)
T
(45)
T_NUM
(24)
Node with value 24 is created and passed upwards to E
T_NUM
(45)
Node with value 45 is created and passed upwards to E

So our grammar would look like this

```
E : E ‘+’ T {$$ = new_node(‘+’);}
   |  T {$$ = $1;}
T: T_NUM {$$ = new_node(45);}
```

Note:

- The different rules in the yacc file will return different types.
- Ensure that YYSTYPE is NOT defined. As this will override any other type definitions you make in the yacc file.
- To define what type a given terminal or non-terminal will return, you will need to create a union. Assuming the return types used are int, char* and an AST struct pointer, the following union can be used:

```
%union
{
   int i;
   char* text;
   struct ast* ast_t;
}
```

Note:

- After defining the union, you will need to define what type (if any) a given terminal or non-terminal will return.
- For example:
    - Assuming the following union is defined:

```
%union
{
    int i;
    char* text;
    struct ast* ast_t;
}
```

    - T_ID is a terminal that returns a struct defined in the union.

```
/* defining the terminal return type */
%token <ast_t> T_ID
```

    - E is a non-terminal that returns the same struct.

```
/* defining the non-terminal return type */
%type <ast_t> e
```

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu