Date: 14/08/2024

Experiment 2.5

AIM

To generate abstract syntax tree of an expression

ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by YACC program.
    2. If input is a sequence of digits:
        1. Copy value to yylval
        2. Return token INTEGER
    3. If input is a sequence of alphabets, digits, or '_' character sucha that it doesn't start with a digit:
        1. Copy value to yylval
        2. Return token IDENTIFIER
    4. If input is an arithmetic operator, parenthesis, semicolon, '=' symbol, or a newline character, return the first character in the input.
    5. For any other symbol, do nothing.
3. Create YACC to parse input as follows:
    1. Create structure AST to represent node of abstract syntax tree with following attributes:
        1. Union val storing value of the node. This can be an integer value, the name of the identifier, or a binary operator.
        2. Variable kind to identify the type of value stored.
        3. Left and right subtree pointers.
    2. Declare following functions:
        1. make_int takes a numerical value as input and returns an AST leaf node with that value.
        2. make_id takes string identifier as input and returns an AST leaf node with that value.

3. make_binop takes an operator and two AST nodes as input and returns a tree with a node representing the operator as root node with the input nodes as children.
4. make_assign takes a string value and an AST node as input and makes node assigning the string as value and the AST node as rightr subtree.
5. print_ast recursively prints a given tree with appropriate indentation.

3. Define parser grammar as follows:
    1. The input consists of multiple statements. Each statement returns an AST which should be printed and deleted after parsing.
    2. Each statement contains one of the following:
        1. IDENTIFIER = expression ;

            In this case, value of the statement can be obtained as make_assign(identifier,expression).

        2. expression ;

            In this case, the value of statement is that of the expression.

    3. Each expression can be expanded as one of the following:
        1. expression + term

            In this case, the value of resultant expression can be obtained as make_binop(‘+’,expression, term).

        2. expression – term

            In this case, the value of resultant expression can be obtained as make_binop(‘-’,expression, term).

        3. term

            In this case, the value of the expression is that of the term.

    4. Each term can be expanded as one of the following:
        1. term * factor

            In this case, the value of resultant term can be obtained as make_binop(‘*’,term, factor).

        2. term / factor

            In this case, the value of resultant term can be obtained as make_binop(‘/’,term, factor).

        3. factor

4. In this case, the value of the term is that of the factor.

5. A factor can be expanded as follows:

    1. A factor can be an IDENTIFIER token. In this case, the value of the factor is obtained by passing the value of the identifier to make_id function.

    2. A factor can be an INTEGER token. In this case, the value of the factor is obtained by passing the value of the identifier to make_int function.

    3. (expression)

        In this case, the value of the factor is that of the expression within the brackets.

4. In user code section:

    1. Define error handling.

    2. Define main function to call yyparse().

4. Use lex command to generate C program.

5. Use yacc to create y.tab,h and y.tab.c

6. Compile and run y.tab.c along with lex program

7. Stop

RESULT

Successfully generated abstract syntax tree with YACC and LEX.