CS39003 Compilers Laboratory, Autumn 2024–2025
Assignment No: 2
Date: 12-Aug-2024

Predictive Parsing

In programming languages like Scheme or LISP, programs are written as a set of lists. A list is an ordered collection of items separated by white spaces (spaces, tabs, new lines) and surrounded by parentheses. Each item in a list can be a scalar (like a variable or a constant or an operator) or again a list. In this assignment, you deal only with arithmetic expressions presented in the list format. For example, the expression $(x_1^2 + x_2^2) / (x_1x_2 + 1)$ can be written as follows.

```
( /
        ( +
                ( *  x1  x1 )
                ( *  x2  x2 )
        )
        ( –
                ( *  x1  x2 )
                –1
        )
)
```

The grammar for such expressions is given below. The terminal symbols are colored orange. EXPR is the start symbol.

EXPR	⟶	**(** OP ARG ARG **)**
OP	⟶	**+** | **–** | ***** | **/** | **%**
ARG	⟶	**ID** | **NUM** | EXPR

Here you deal only with binary operators (although Scheme and LISP support any number of arguments for associative operators like + or *). Assume that each number is an integer. Allow negative integers. That is, a NUM is a non-empty sequence of digits optionally preceded by a sign (+ or –). However, do not use the unary minus before variables (as in –x1). Finally, follow the same naming convention for ID's as in C. This grammar is LL(1).

Part 1: Lexical analysis

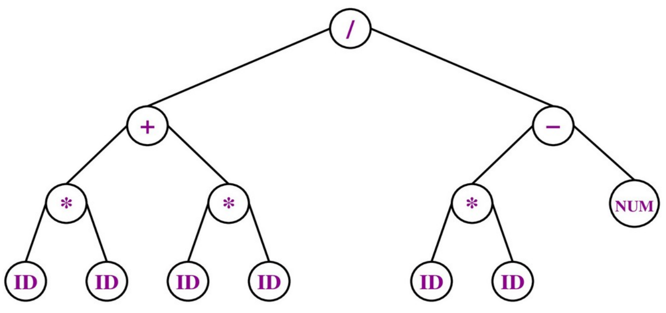
Write a lex program to find out the tokens in the input program. Write a C/C++ program that stores the ID's in a table T (the symbol table) and the NUM's in another table C (the table of constants). Implement your own data structures for these tables.

Part 2: Table for predictive parsing

Manually prepare the predictive parser table M(A,a), where A can be EXPR, OP, or ARG, and a can be (,), +, –, *, /, %, ID, and NUM. You do not have to write a program to build the table. Work out the entries using pen and paper or in your head.

Part 3: Predictive parsing for LL(1) grammars

Implement the parsing algorithm using the table M and a stack S. You may use the STL stack data structure. But implement your own data structure for the parse tree. For the above example, the parse tree would look as in the adjacent figure. Each internal node in the tree stores an operator, and has exactly two child nodes. Each leaf node stores an operand. If the operand is an ID, then the leaf should store a reference to an appropriate entry in T. If the operand is a NUM, then the leaf should store a reference to an appropriate entry in C. Clearly, the parse tree can be implemented by a binary tree. Each node in the tree should



store (i) a type (OP/ID/NUM), (ii) a union of an operator, a reference to an entry in T, and a reference to an entry in C, and (iii) two child pointers. If needed, you may additionally include parent pointers in the nodes. After the parsing process completes successfully, print the parse tree in the format specified in the sample output below.

Part 4: Evaluate the expression

The formula may contain variables (ID’s). Although Scheme/LISP has constructs to declare and initialize variables, you do not have to implement these constructs in this assignment. Instead the input file should contain integer values for all the variables appearing in the expression, after the expression is complete. These values should populate the variables in the order of their first occurrences in the formula. Run though the table T. For each ID appearing in T, the value read from the input file should be used to initialize that variable. These values would be stored in the appropriate entries in T itself. A variable may appear multiple times in the expression, but you read the value of each variable only once. After all the variables have got their values, evaluate the expression by recursively traversing the parse tree. Finally, print the value of the expression.

Part 5: Detecting errors

Detect errors during the building of the parse tree. The errors may include invalid operators and invalid numbers of arguments to operators. You do not have to correct the errors. Just reporting the error followed by premature termination will do. A real-life compiler (or interpreter) should report the exact error location (line number + position in line) too, but you do not have to do this in this assignment.

An error is detected whenever the stack top A does not tally with the next input symbol a, that is, when M(A,a) is empty. For example, suppose that EXPR at the stack top is replaced by (OP ARG ARG). Subsequently, (is matched, and the two subtrees corresponding to the two ARG’s are created. The stack now has) at the top, so the next input symbol must be). If the next token from the input is anything else, an error is detected. The sample outputs show a few types of errors that can be detected this way.

What to submit

Submit two files: your lex program list.l and your C/C++ program evalexpr.c (or evalexpr.cpp). Your C/C++ program will do all the jobs including getting tokens, building and printing the parse tree, getting user inputs, evaluating the parse tree, and printing the final result.

Sample outputs

Input file

```
( /
   ( +
       ( *  x1  x1 )
       ( *  x2  x2 )
   )
   ( -
       ( *  x1  x2 )
       -1
   )
)
-125
16
```

Output

```
Parsing is successful
OP(/)
---> OP(+)
      ---> OP(*)
            ---> ID(x1)
            ---> ID(x1)
      ---> OP(*)
            ---> ID(x2)
            ---> ID(x2)
---> OP(-)
      ---> OP(*)
            ---> ID(x1)
            ---> ID(x2)
      ---> NUM(-1)
Reading variable values from the input
x1 = -125
x2 = 16
The expression evaluates to -7
```

Input file

```
( +
 4
 ( +
   ( *
     ( +
       ( *
         14
         (+ 6 (+ 12 (* 2 16)))
       )
       9
     )
     ( *
       (+ (* 11 ( + 3 21)) (* 15 (* 8 13)))
       19
     )
   )
   ( + (* 5 18)
     (+
        (+ 1 (* 10 (+ 7 17)))
        20
     )
   )
 )
)
```

Output

```
Parsing is successful
OP(+)
---> NUM(4)
---> OP(+)
      ---> OP(*)
            ---> OP(+)
                  ---> OP(*)
                        ---> NUM(14)
                        ---> OP(+)
                              ---> NUM(6)
                              ---> OP(+)
                                    ---> NUM(12)
                                    ---> OP(*)
                                          ---> NUM(2)
                                          ---> NUM(16)
                  ---> NUM(9)
            ---> OP(*)
                  ---> OP(+)
                        ---> OP(*)
                              ---> NUM(11)
                              ---> OP(+)
                                    ---> NUM(3)
                                    ---> NUM(21)
                        ---> OP(*)
                              ---> NUM(15)
                              ---> OP(*)
                                    ---> NUM(8)
                                    ---> NUM(13)
                  ---> NUM(19)
      ---> OP(+)
            ---> OP(*)
                  ---> NUM(5)
                  ---> NUM(18)
            ---> OP(+)
                  ---> OP(+)
                        ---> NUM(1)
                        ---> OP(*)
                              ---> NUM(10)
                              ---> OP(+)
                                    ---> NUM(7)
                                    ---> NUM(17)
                  ---> NUM(20)
The expression evaluates to 24571459
```

Input file	Output

`(+ (* a b c) d)`
`5 6 7 8`
	`*** Error: Right parenthesis expected in place of c`

`(% (- x (+ y (/ z 5)) (* a b))`
`1 -2 3 -4 5`
	`*** Error: Right parenthesis expected in place of (`

`(+ a (* (% c d)))`
`-2 -3 -4`
	`*** Error: ID/NUM/LP expected in place of )`

`(+ (* a b) (+ c *))`
`8 9 10`
	`*** Error: ID/NUM/LP expected in place of *`

`(+ (* a (&& b c)))`
`20 15 10`
	`*** Error: Invalid operator && found`

`(+ (* a b) (x * y))`
`1 0 -1 5`
	`*** Error: Operator expected in place of x`