CS39003 Compilers Laboratory, Autumn 2024–2025

Lab Test 1

30–August–2024 | 07:15pm–08:15pm | Maximum marks: 50

Roll no: _____ **Name:** _____

Write in the respective spaces provided. Write syntactically correct codes (no credits for pseudocodes).

Consider arithmetic expressions in the list notation with only two operators + and *. Allow each operator to take any positive number of arguments. Assume that the expressions contain no variables, and each numeric operand in expressions is a floating-point number (with or without a fractional part and/or an exponent part). Your task in this test is to develop a **predictive parser** to build trees for such expressions.

Define our language of expressions using the following LL(1) grammar. Here, EXPR is the start symbol. The other nonterminal symbols are OP, ARG, and REST. The terminal symbols are (,), +, *, and *Num*.

$$
\begin{aligned}
\text{EXPR} &\rightarrow (\ \text{OP}\ \ \text{ARG}\ \ \text{REST}\ ) \\
\text{OP} &\rightarrow + \mid * \\
\text{ARG} &\rightarrow \textit{Num} \mid \text{EXPR} \\
\text{REST} &\rightarrow \text{ARG}\ \ \text{REST} \mid \varepsilon
\end{aligned}
$$

1. Write your own lex file for recognizing the tokens from an expression in the list format, and to discard all unusable characters (white spaces and invalid characters) from the input. Macros for all the grammar symbols are supplied at the beginning of the lex code. In what follows, use these macros. Note that lex should return only the tokens for the terminal symbols. The macros for nonterminals will be used in Part 4. **(8)**

If your answer does not fit in the first column,	*continue to the second column.*

`%{`
`#define EXPR 1001`
`#define OP 1002`
`#define ARG 1003`
`#define REST 1004`
`#define LP 1005`
`#define RP 1006`
`#define PLUS 1007`
`#define STAR 1008`
`#define NUM 1009`
`%}`
	`ws  [ \t\n]+`
	`digits  [0-9]+`
	`frac  \.{digits}`
	`expt  [eE][+-]?{digits}`
	`num  [+-]?{digits}{frac}?{expt}?`
	
	`%%`
	
	`{ws}  { }`
	`{num}  { return NUM; }`
	`"("  { return LP; }`
	`")"  { return RP; }`
	`"+"  { return PLUS; }`
	`"*"  { return STAR; }`
	`.  { }`
	
	`%%`

2. In the table below, derive the FIRST and FOLLOW values for the nonterminal symbols of the given expression grammar. Supply brief justification for each entry. **(8)**

Name	Value	Justification
FIRST(EXPR)	{ (}	The only rule for EXPR is EXPR → (OP ARG REST)
FIRST(OP)	{ +, * }	We have the only rules OP → + | *
FIRST(ARG)	{ *Num*, (}	FIRST(ARG) = FIRST(*Num*) ∪ FIRST(EXPR)
FIRST(REST)	{ *Num*, (, ε }	FIRST(REST) = FIRST(ARG) ∪ {ε}
FOLLOW(EXPR)	{ *Num*, (,), $}	FOLLOW(EXPR) should contain $ (because EXPR is the start symbol), and all symbols of FOLLOW(ARG) (because of the rule ARG → EXPR)
FOLLOW(OP)	{ *Num*, (}	OP is only followed by ARG (in the production EXPR → (OP ARG REST)), so FOLLOW(OP) = FIRST(ARG)
FOLLOW(ARG)	{ *Num*, (,) }	ARG is only followed by REST, so all terminal symbols in FIRST(REST) will be in FOLLOW(ARG). Moreover, REST can be empty, so we put all of FOLLOW(REST) in FOLLOW(ARG) (because of REST → ARG REST)
FOLLOW(REST)	{) }	For the only applicable rule EXPR → (OP ARG REST)

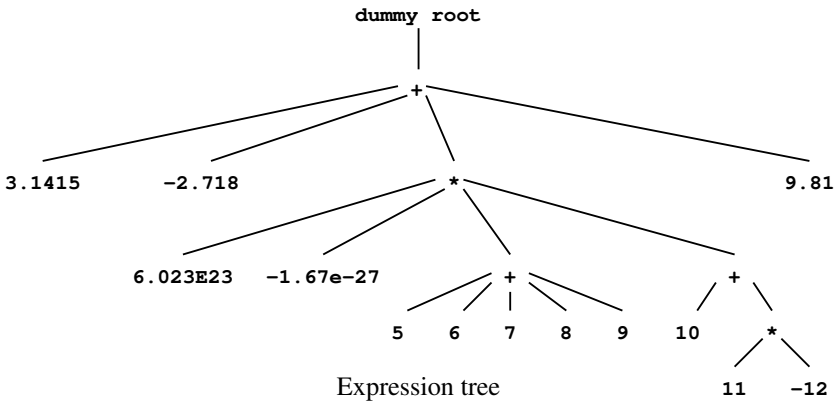
3. Using your answer of Part 2, populate the **LL(1) parsing table** below, for the **predictive parser**. Here, we have no need to consider the end-of-input marker $ (so long as the input is a single valid expression). **(8)**

Non-terminal	Input symbol				
	+	*	(	)	*Num*
EXPR			EXPR → (OP ARG REST)		
OP	OP → +	OP → *			
ARG			ARG → EXPR		ARG → *Num*
REST			REST → ARG REST	REST → ε	REST → ARG REST

4. You need to write a function `parse()` to return an expression tree for an input expression in the list format. An example of an expression and the corresponding tree is shown in the figure below. The numeric operands are real-valued, and are stored in the tree itself (that is, no separate table for constants is maintained).

```
( +
    3.1415
    -2.718
    ( *
        6.023E23
        -1.67e-27
        ( + 5 6 7 8 9 )
        ( + 10 (* 11 -12) )
    )
    9.81
)
```

Expression



Expression tree

The parsing algorithm uses three data structures explained in the table below. You do not need to implement these data structures. Use only the functions given against the data structures.

Data structure	Explanation
`exprtree`	The data type for a node of the tree is `et_node`. The initialization function is `et_init()`. A function `addchild(et_node *N, int t)` creates a new child of the node `N` of the tree, and returns a pointer to this new child. `t` is the type of the node (OP or NUM).
`parsestack`	This is a stack of integers used to store the grammar symbols during parsing. Use the integer codes (macros) of Part 1 for the grammar symbols (terminals and nonterminals). This stack has its usual `init`, `empty`, `top`, `push`, and `pop` operations.
`nodestack`	In order to keep track of the current expression-tree node `N` whose child nodes are created by `addchild()`, maintain a stack of these nodes. The top of this stack is the node used in each call of `addchild()`. This stack supports `init`, `empty`, `top`, `push`, and `pop`.

Fill in the blanks in the following function `parse()` using appropriate C/C++ code, in order to implement the **predictive parsing algorithm** under the **given LL(1) grammar**. No need to detect errors in input.

```
exprtree parse ( )
{
    exprtree   ET = et_init();          /* Initialize to empty tree with a dummy root */
    parsestack PS = ps_init();                    /* Initialize to empty parse stack */
    nodestack  NS = ns_init(ET);           /* Initialize node stack by the dummy root */
    int A, /* A is the grammar symbol at the top of PS */
        a; /* a is the next input symbol (token returned by lex) */

    ____________ PS = ps_push(PS, EXPR); ____________ /* Prepare PS for parsing */  (1)

    a = ____________ yylex(); ____________ /* Read the first input token */  (1)

    while ( ____________ !empty(PS) ____________ ) {                                (1)

        ____________ A = ps_top(PS); PS = ps_pop(PS); ____________ /* Extract A from PS */  (1)
        switch (A) {
            case EXPR:                                                             (4)

                PS = ps_push(PS,RP);
                PS = ps_push(PS,REST);
                PS = ps_push(PS,ARG);
                PS = ps_push(PS,OP);
                PS = ps_push(PS,LP);
                break;
```

```
        case OP:                                                    (4)

            if (a == PLUS) PS = ps_push(PS, PLUS);
            else if (a == STAR) PS = ps_push(PS, STAR);
            break;

        case ARG:                                                   (4)

            if (a == LP) PS = ps_push(PS,EXPR);
            else if (a == NUM) PS = ps_push(PS,NUM);
            break;

        case REST:                                                  (4)

            if ((a == LP) || (a == NUM)) {
                PS = ps_push(PS,REST);
                PS = ps_push(PS,ARG);
            } else if (a == RP) {
                NS = ns_pop(NS);
            }
            break;

        /* Handle the cases of terminal symbols */                  (6)

        case PLUS:
        case STAR:
        case NUM:
        case LP:
        case RP:
            if ( (a == PLUS) || (a == STAR) ) {
                NS = ns_push(NS, et_addchild(ns_top(NS), OP));
            } else if (a == NUM) {
                et_addchild(ns_top(NS), NUM);
            }
            a = yylex();
            break;

    }
  }

  return ET;
}
```