CS 435 Project 2

Recursive Descent Parser (with Scanner) for Simple_PL1

Overview of Assignment

- Using your Project 1 scanner, implement a recursive descent parser in C for Simple_PL1.
- If a lexical error or parser error is encountered, report the error according to the specification shown below and exit.

Review of *Simple_PL1 Lexical Specification*

Your scanner must recognize the following tokens. For definitions below: `letter = [a..zA..Z]` and `digit = [0..9]`.

Token Name (used internally by compiler)	Regular expression definition	Lexeme examples
ID	`(letter|_)(letter|digit|_)*`	max, a_1, _a1,
READ	`read`	note: `read` is an identifier, but is **reserved** and needs to be returned from scanner as distinct token.
WRITE	`write`	note: `write` is an identifier, but is **reserved** and needs to be returned from scanner as distinct token.
NUMBER	`digit digit*`	1, 21, 1024
ASSIGN	`:=`	
PLUS	`+`	
MINUS	`-`	
TIMES	`*`	
DIV	`/`	
SEMICOLON	`;`	
COMMA	`,`	
LPAREN	`(`	
RPAREN	`)`	
SCAN_EOF		scanner returns this when the source end-of-file is reached.

WHAT TO SUBMIT: Name your project CS435P02<YourLastName>. After completing the project, zip the entire project folder and test cases. Name your zip file CS435P02<YourLastName>.

I will grade your project by running it against a collection of test files. (The executable file name for *this example* is `CS435P02Jeffrey`. **Your** executable will be your project name.)

Example: The command line should be: `$ CS435P02Jeffrey src1.txt`

TO DO: This parser must be written in C using the Visual Studio 2022 IDE. Utilize your scanner from Project 1. You must use the command-line parameters to access

the source file, i.e. argv[1]. Open the source file and parse it. You must use recursive descent parsing. For this simple parser, when a parsing error occurs, just print the expected token (use mnemonic array of strings), or, indicate that an unexpected token was found and exit the program. You will need to analyze which message is appropriate. When a lexical error occurs, indicate that an unrecognized character was seen. Do not attempt error recovery.

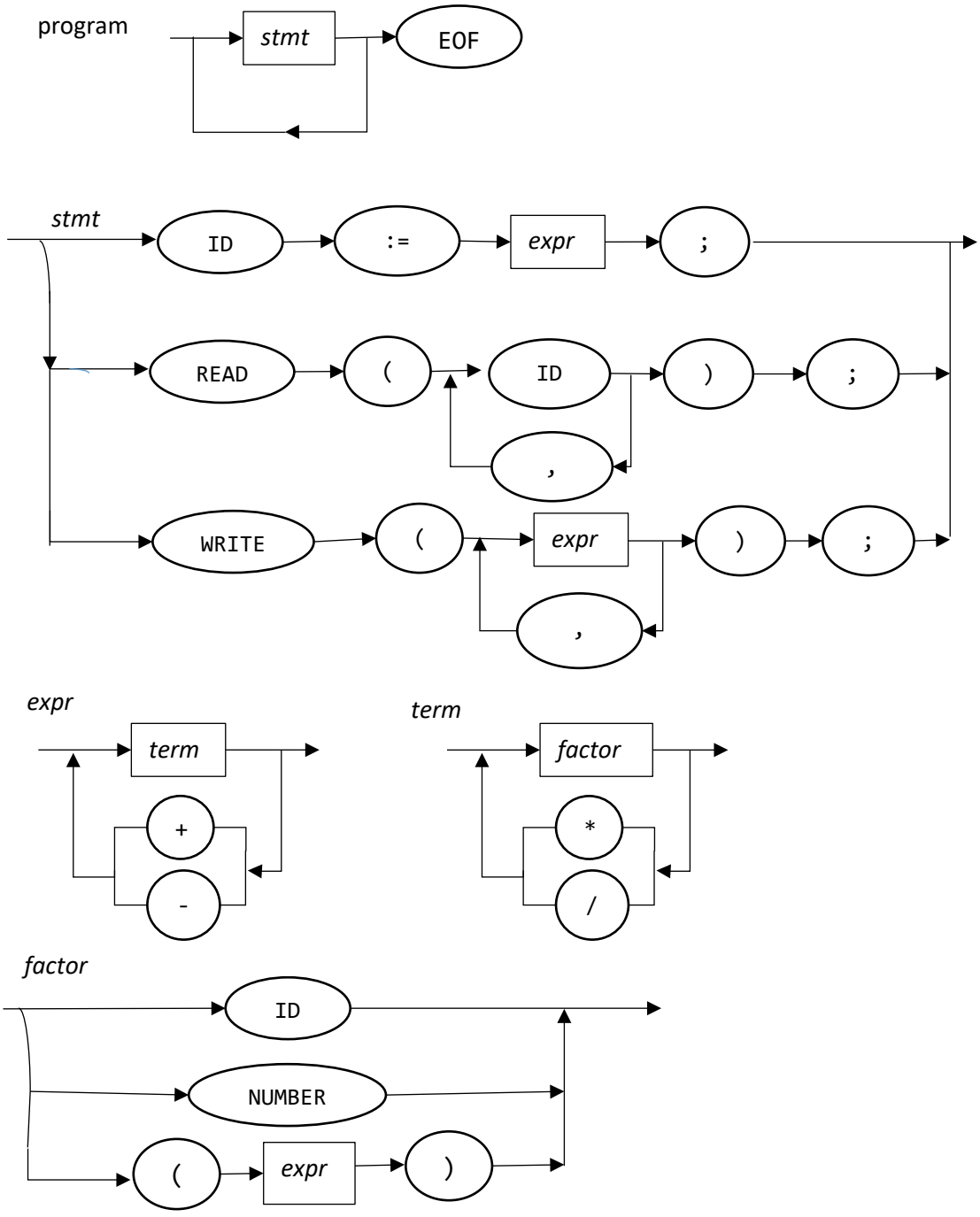
Simple_PL1 Syntax Definition and Grammar:

The Simple_Pl1 programming language is defined and is generated by the grammar shown below. (The "$" represents the SCAN_EOF token.) The recursive descent parser must be based upon the following grammar, which meets the conditions for building a recursive descent parser. You can also alter it to make it more concise and for building a more efficient recursive descent parser. For example, see the syntax diagram on the next page.

Note that you can take advantage of checking whether the next token is in the first set of a particular non-terminal when parsing a production rule for that non-terminal.

program	→	*stmt_list* `$`
stmt_list	→	*stmt stmt_list* | ε
stmt	→	`id :=` *expr*`;` | `read(` *id_list* `);` | `write(` *expr_list* `);`
expr_list	→	*expr expr_list_tail*
expr_list_tail	→	`,` *expr expr_list_tail* | ε
id_list	→	`id` *id_list_tail*
id_list_tail	→	`,` `id` *id_list_tail* | ε
expr	→	*term term_tail*
term_tail	→	*add_op term term_tail* |ε
term	→	*factor factor_tail*
factor_tail	→	*mult_op factor factor_tail* | ε
factor	→	`(` *expr* `)` | `id` | `number`
add_op	→	`+` | `-`
mult_op	→	`*` | `/`

Syntax Diagram: This syntax diagram expresses an alternative presentation of a grammar. This grammar is a revision of the original grammar shown above and generates the same Simple_PL1 language. This grammar shown below can aid in the engineering of a recursive descent parser.



Handling Syntax Errors in Simple_PL1 source code.

If there are no parsing errors, print to standard output the following message:

```
Parsing complete. No errors.
```

If there is a detected parsing (or lexical error), print the token name that was going to be or was recognized and exit the program. Use the following format for a parser error:

```
Expected symbol: <a token name> or Unexpected symbol: <a token name>.
```

You can also use the first sets to detect an error inside a certain construct, such as expressions. Use an error message *such as* (just an example):

```
Error in expression: Expected ID, NUMBER, or '('.
```

The expected or unexpected symbol should be a token nearby the actual parsing error. Note: sometimes the error might appear to be misleading. Error reporting and recovery is hard. Utilize the mnemonic array of c-strings from Project 1 to print the token name.

Shown below are some examples.

	Source File		Standard output

Example 1	`x := 2;`
	`y := 3;`
	`read(a, b);`
	`write(a,b,a+b*(2*x/y));`
		⇒	`Parsing complete. No errors.`

Example 2	`x := 2;`
	`y := 3;`
	`read(a, b);`
	`write(a,b,a+b*(2*x/y;`
		⇒	`Expected symbol: RPAREN`

Example 3	`x := 2;`
	`y := 3;`
	`read(a, b)`
	`write(a, b,a+b*(2*x/y));`
		⇒	`Expected symbol: SEMICOLON`

Example 4	`x := 2;`
	`y := 3;`
	`read(a b);`
	`write(a, b,a+b*(2*x/y));`
		⇒	`Expected symbol: RPAREN`

Example 5	`read(x, y, z, v);`
	`read( c ); temp := (x -`
	`y)*(2*x + z/4) /v;`
	`a :=  (c);`
	`write( temp /    );`
		⇒	`Error in expression: Expected ID, NUMBER, or '('.`

NOTES:

In Example 2, there is a missing right parenthesis in the last statement.

In Example 3, there is a missing semicolon that should terminate the read statement on the third line.

In Example 4, there is a missing comma. Since a comma separates identifiers in an identifier list for READ, the parser "thinks" it is at the end of the list when no comma is found and expects a RPAREN. Perhaps the message should be "`missing comma`," but adding code to detect this kind of error would complicate the parser. Although this is a simple fix in this case, this demonstrates the difficulty of reporting errors and the design decisions compiler engineers face.

In Example 5, you can see how a specific construct can be referenced in the error message. The first(Expr) set can be helpful for detecting and reporting this kind of error.

Implementing a Recursive Descent Parser for Simple_PL1 and Some Supporting Functions: Utilize the textbook as a guide for writing your recursive descent parser. Also, shown below are two helpful functions for supporting your recursive descent parser and implementing error reporting, such as in the examples shown above.

```
void parse_error(char *errMsg, char *lexeme) {
	extern unsigned numErrs; //for future if error recovery used
	numErrs++;
	fprintf(stderr, "%s: %s\n", errMsg, lexeme);
}

void match(enum tokenType expected)
{
	if (currentToken == expected) {
		currentToken = scan();
	}
	else{
		parse_error("Expected symbol", mnemonic[expected]);
		exit(1);
	}
}
```