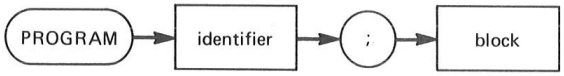
CSE 4714 / 6714 — Programming Languages
Project Part 2

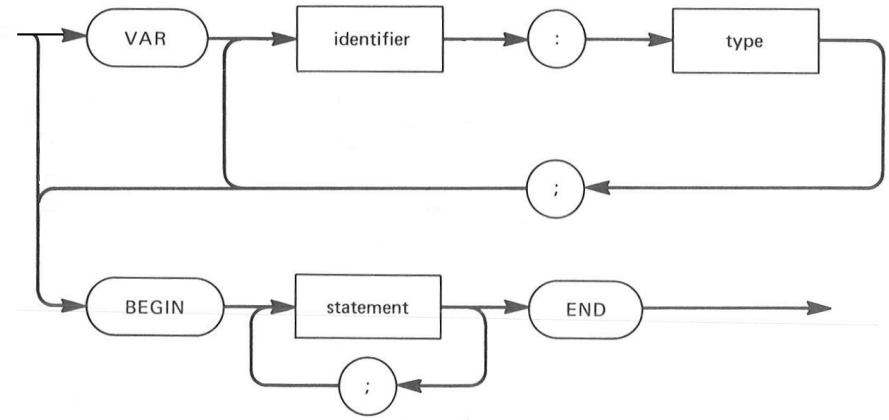
Create a recursive descent parser for a subset of Ten Instruction Pascal Subset (TIPS). Use your corrected `rules.l` from Part 1 for the lexical analyzer portion of your parser.

The portion of TIPS that your parser must recognize:

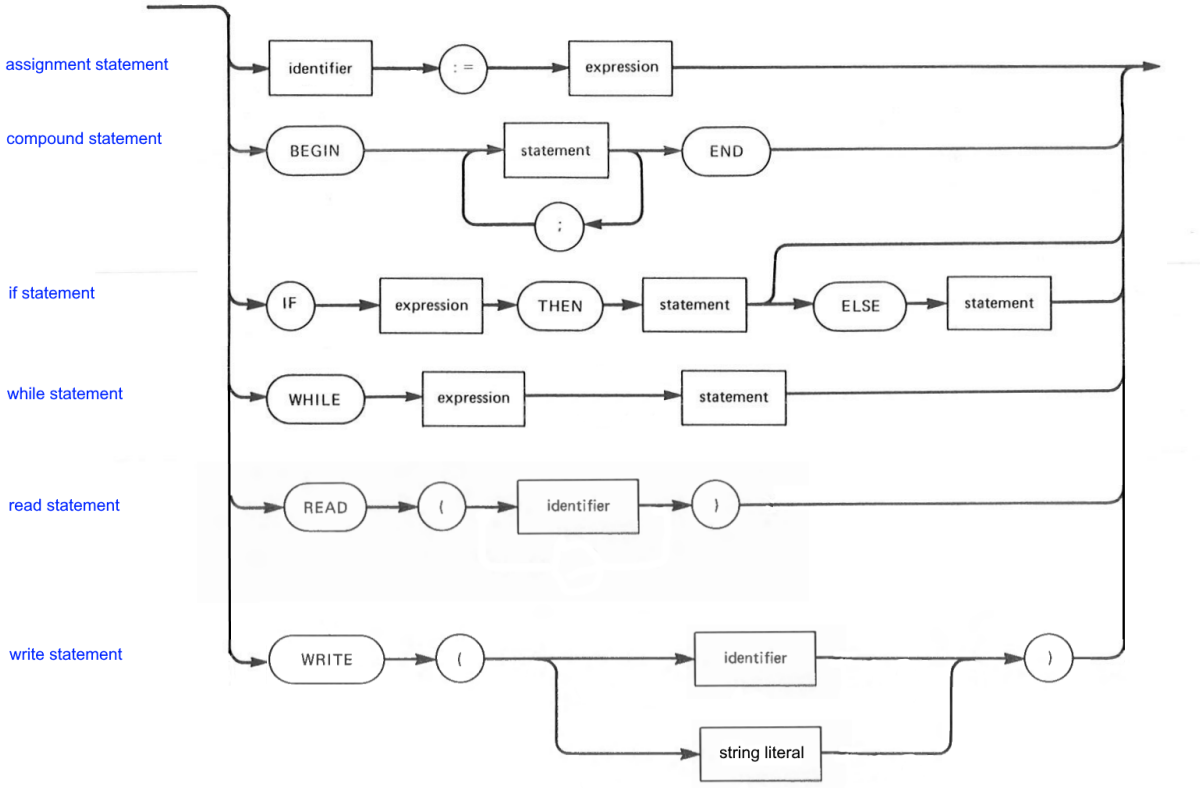
program



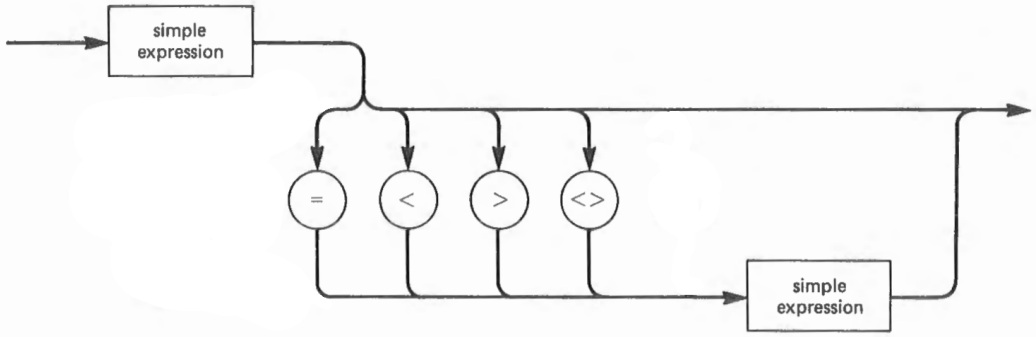
block



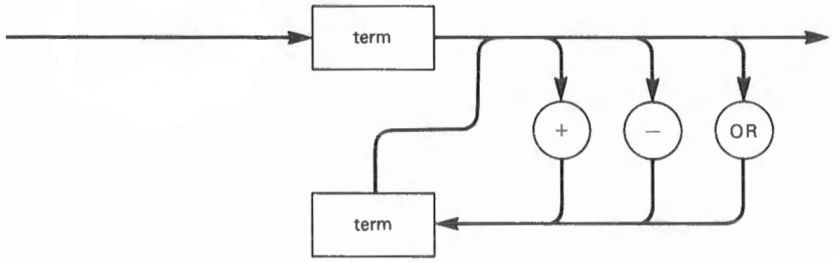
statement



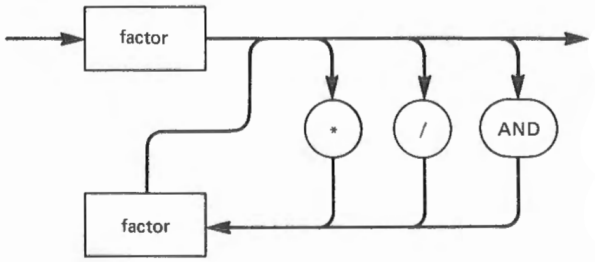
expression



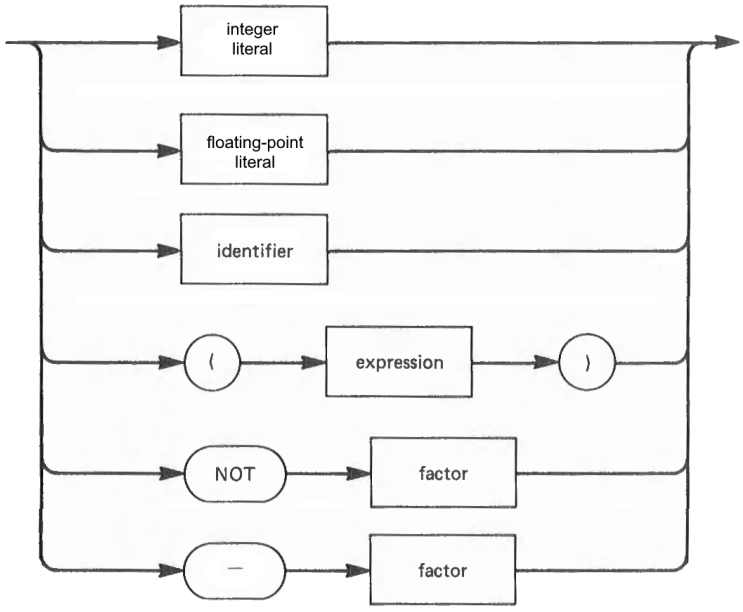
simple expression



term



factor



A great place to start is to translate the previous syntax diagrams into EBNF notation. Then calculate the 'First Token Set' for each of the production rules. Writing a recursive descent parser is basically writing a function that recognizes each of the production rules.

While any programming language will have EBNF rules (and therefore parsing functions) for standard arithmetic constructs such as terms and factors, those rules do not always exactly match. While the code discussed as part of Lecture 5 (`recursive-descent-parser-example.zip`; posted on Canvas) is a good example of a recursive descent parser for an arithmetic expression, please do not expect to directly use its code in your parser for the TIPS programming language.

Grammar Productions

<program> → TOK_PROGRAM TOK_IDENT TOK_SEMICOLON <block> First Token Set: { TOK_PROGRAM }
<block> → [TOK_VAR TOK_IDENT TOK_COLON (TOK_INTEGER | TOK_REAL) TOK_SEMICOLON { TOK_IDENT TOK_COLON (TOK_INTEGER | TOK_REAL) TOK_SEMICOLON }] <compound> First Token Set: { TOK_VAR, TOK_BEGIN }
<statement> → <assignment> | <compound> | <if> | <while> | <read> | <write> First Token Set: {TOK_IDENT, TOK_BEGIN, TOK_IF, TOK_WHILE, TOK_READ, TOK_WRITE}
<assignment> → TOK_IDENT TOK_ASSIGN <expression> First Token Set: { TOK_IDENT }
<compound> → ... First Token Set: ...

Note: Symbols shown in red are EBNF symbols, not actual symbols expected.

Syntax errors are found in two ways:

- If a parsing function is called and the next input token is not a member of its first token set, a syntax error has been found. For example, if you start parsing a TIPS program and the very first token is not TOK_PROGRAM, you found a syntax error and the input program is not syntactically correct.
- If the next input token is not what is expected at any point in the parsing of the input program, a syntax error has been found. For example, if your program starts parsing a <block> and finds that the first token is TOK_VAR, the very next token has to be a TOK_IDENT. If the next token is anything else, you found a syntax error.

Your parser should print out a listing of the input TIPS program (indented to simulate a parse tree) along with a **symbol table**. The symbol table will consist of a list of variables created / used in the parsed input. Sample output for both a correct and incorrect TIPS program will be provided.

If an error is found in the input (a program written in TIPS), the parser should halt.

- If a syntax error is found, an error message listing the line number, the last lexeme read from the input, and the specific error should be displayed. A list of expected error messages is available in `error_messages.txt`.
- If a variable is declared twice, an error message should be displayed.
- If a variable is used before it has been declared within a `<block>` statement, an error message should be displayed.

Use the files in `Part_2_Starting_Point.zip` as a starting point for your parser. Edit the headers in the files to provide your name and the purpose of the files.

The deliverable is a zip file of the files needed to build and execute your parser (`makefile`, `rules.l`, `productions.h`, `driver.cpp`, etc.). Create a zip file named `netid_part_2.zip` and upload that file to the assignment. Do not include any files generated by the `makefile` in your submission. For example, your submission should not include any `.o` or `.exe` files.

Hint

Do not try to write the complete parser before starting to debug your code. Start with the easier statements first, such as `<read>` and `<write>`. When starting to work on `<expression>` use a much simpler rule to begin with such as:

```
<expression> → TOK_INTLIT
```

Then add the additional rules after confirming that the simplified versions work as expected.

Store the variable names in a `set` datatype:

```
extern set<string> symbolTable;
```