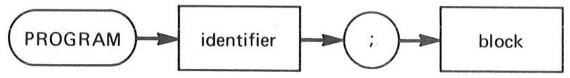
CSE 4714 / 6714 — Programming Languages
Project Part 2

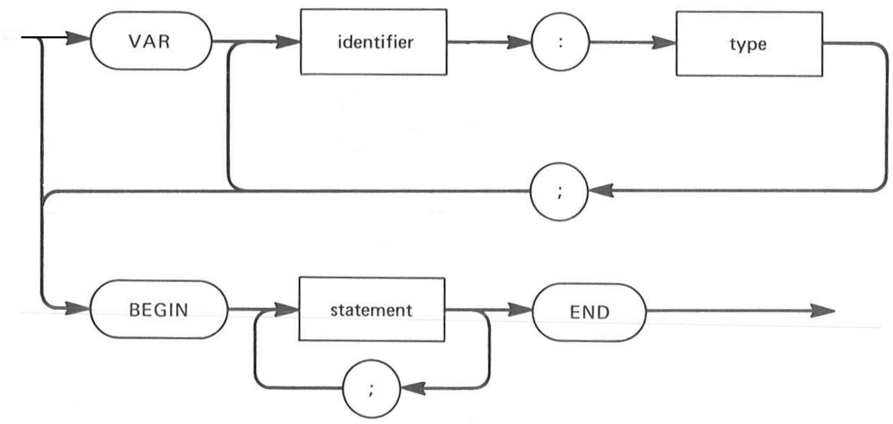
Create a recursive descent parser for a subset of Ten Instruction Pascal Subset (TIPS). Use your corrected `rules.l` from Part 1 for the lexical analyzer portion of your parser.

The portion of TIPS that your parser must recognize:

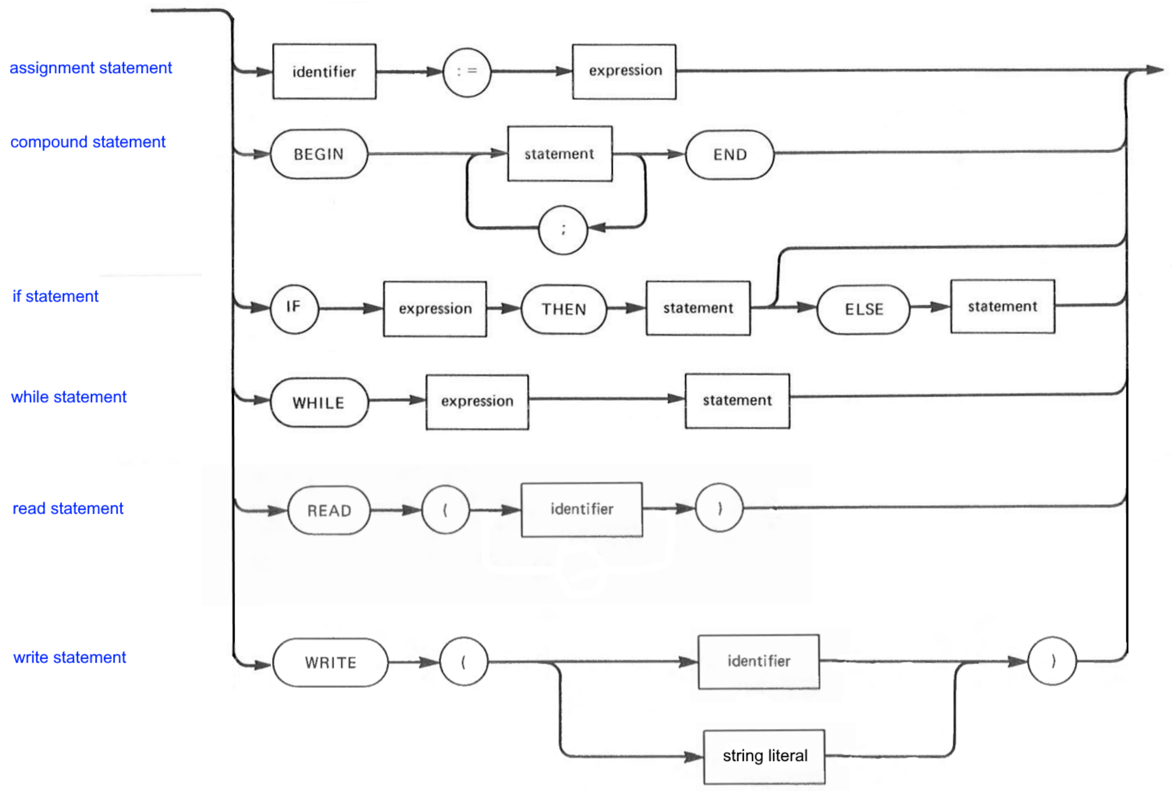
program



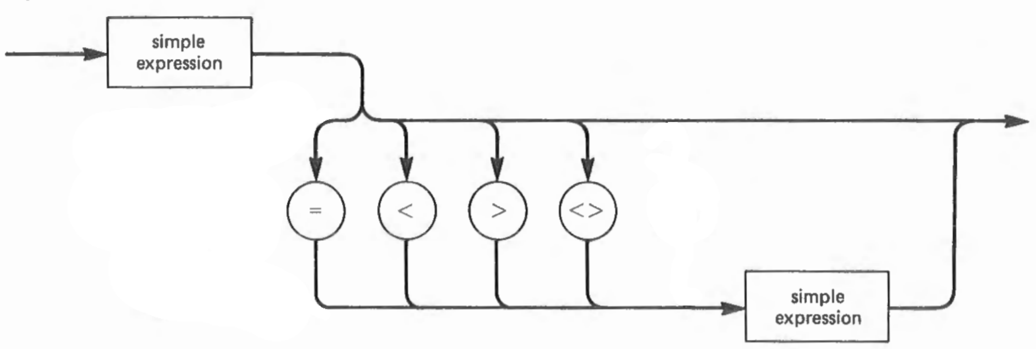
block



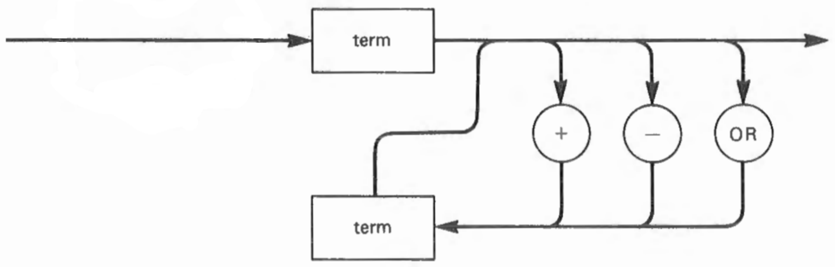
statement



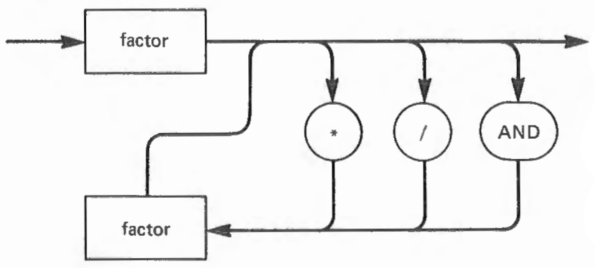
expression



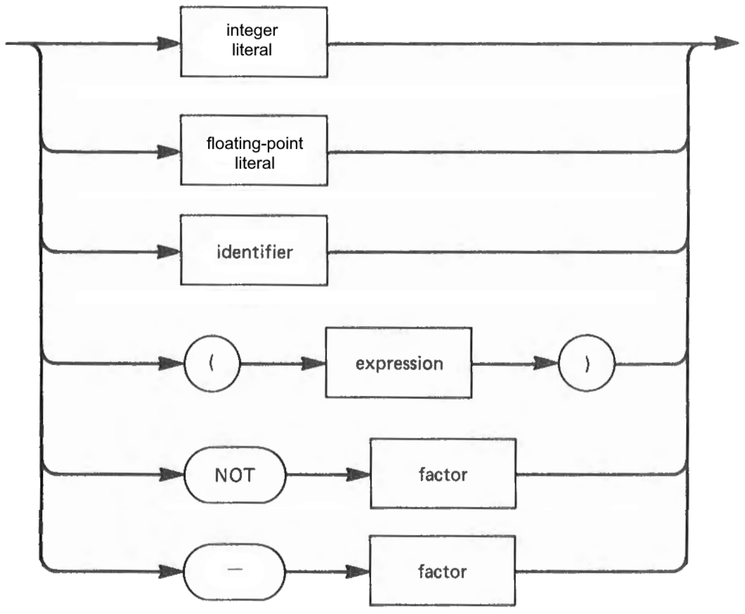
simple expression



term



factor



A great place to start is to translate these 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> => PROGRAM IDENTIFIER ; <block>

First Token Set: { TOK_PROGRAM }
```

```
<block> => [ VAR IDENTIFIER : ( INTEGER | REAL ) ;
               { IDENTIFIER : ( INTEGER | REAL ) ; } ]
           <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> => IDINTIFIER := <expression>

First Token Set: { TOK_IDENT }
```

```
<compound> => ...

First Token Set: ...
```

Note: Symbols shown in **`bold red`** are EBNF metasymbols, not the 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, when first parsing a TIPS program, if the very first token is not `TOK_PROGRAM`, that is a syntax error. In this case, the correct behavior is to `throw` with the error message shown in `error_messages.txt`, `"3: 'PROGRAM' expected"`. In the file `driver.cpp`, the `catch` statement prints this message, along with some additional error information, and then quits.
- 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, when parsing a `<block>`, if the next token is `TOK_VAR`, the very next token must be `TOK_IDENT`. If the next token is anything else, it is a syntax error. `error_messages.txt` defines the error message for this situation as well.

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 and used in the parsed input. Sample output for both correct and incorrect TIPS programs are provided.

As discussed above, 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. This is defined in the `catch` statement in `driver.cpp`. The list of expected error messages is 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.

Written Report

Write a report about your experience. The report should cover the following items:

1. Describe, in a few sentences, the development process you used during this assignment. In what order did you develop the parsing functions? How many functions did you write?
2. As you wrote the code to implement each production (`<program>`, `<block>`, `<statement>`, etc.), you should have used one of the syntax diagrams above to create a grammar production (EBNF rule), similar to the ones shown above. Choose at least three of these grammar productions, and include them in your report.
3. Again discuss the role of the program call stack in the parsing process. Which one of the test input programs resulted in the deepest call stack? Why was it this deep?
4. Describe, in a few sentences, any sources you consulted during your development of this assignment.
5. In addition to the requirements specified above, the report should be neat, well structured, grammatically correct, and use words that are spelled correctly.

List your *name*, *netID*, and the *date* at the top of your report. Otherwise, the report may use any format and fonts. The report does not have any specific length requirement. Completeness is valued more than length, and excessive length could result in a lower grade.

Written Report Grading Rubric

Criteria	Ratings				Points
Development Process	**rank 1: Mastery** Engaging sentences. Correct and clear writing.	**rank 2: Good** Adequate sentences. Presentation covers material but has minor writing or structural errors.	**rank 3: Developing** Sentences leave out some minor material or are poorly structured. Recurring writing issues. Details not as in depth as required. Poor formatting.	**rank 4: Lacking** No sentences or significant issues. Incomplete, too short or too long, or repeated, significant writing issues. An obvious slap-dash effort.	25
Grammar Productions	**rank 1: Mastery** Productions given, formatting clear and easy to read. Metasymbols are clearly indicated. Productions are correct.	**rank 2: Good** Productions given and are correct, but have minor formatting or structural errors.	**rank 3: Developing** Productions are not correct, or have major formatting issues. Metasymbols are not indicated. Not enough productions are given. Productions are not in proper EBNF format. Details not as in depth as required.	**rank 4: Lacking** No productions or significant issues. An obvious slap-dash effort.	25
Program Stack	**rank 1: Mastery** Engaging answer. Correct and clear writing. Complete answers.	**rank 2: Good** Adequate answer. Presentation covers material but has minor writing or structural errors.	**rank 3: Developing** Answers leave out some minor material or are poorly structured. Recurring writing issues. Details not as in depth as required. Poor formatting.	**rank 4: Lacking** No answers or significant issues. Incomplete, too short or long, or repeated, significant writing issues. An obvious slap-dash effort.	25
Overall Quality	**rank 1: Mastery** Clearly formatted, contains all required elements, attractively laid out.	**rank 2: Good** Some minor formatting issues or missing elements, but report is still easy to follow.	**rank 3: Developing** Significant issues with formatting, or many missing elements. Report is not so easy to follow.	**rank 4: Lacking** No demonstration of mastery. Significant formatting issues, or many missing elements. An obvious slap-dash effort.	25

Deliverable

The deliverable is a zip file of the files needed to build and execute your parser (`makefile`, `lexer.h`, `rules.l`, `parser.h`, `parser.cpp`, `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.

Feel free to assist other students on this lab assignment. If you need additional help, please contact the TA or attend their office hours.

The sample input / output files provided with the assignment are a good starting point for testing your program. You are responsible for testing your program so that the lexical analyzer will function as expected with ***any*** input.

Hints

1. The most important hint is given first: *Do not try to write the complete parser before starting to debug your code!* This strategy will result in a world of pain. Develop one progression at a time, and test it.

2. For developing the productions, here is a suggested order. First, work bottom-up from `<factor>` to `<expression>`.

   Begin by writing a production for `<factor>`. Test cases should be different factors, such as:

   ```
   34
   44.3
   TEST
   ( 21 )
   NOT TEST
   - SUM
   ```

   During the development of `<factor>`, `<expression>` should be a production that uses a much simpler rule, such as `<expression> → INTLIT`.

   Next, develop a production for `<term>`. Follow this with a `<simple_expression>`, and finally an `<expression>`. Thoroughly test every possibility before moving on to the next production.

   Once `<expressions>` have been implemented, start at `<program>` and work top-down. Start with the easier productions first, such as `<read>` and `<write>`.

3. Store the variable names in a `set` datatype:

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

4. The `diff` command discussed in Lab 2 will continue to be useful.

   The general command to invoke the parser looks like this (invoked from within the `Part-2-Test_Cases` directory):

   ```
   $ ../Part-2-Starting_Point/tips_parse 1-hello.pas
   ```

   By default, everything the parser does results in a print statement. While this can help with debugging and understanding the code flow, it also results in a large amount of output. The following modification removes the lines that contain the word *found*, and therefore better reveal the tree structure:

   ```
   $ ../Part-2-Starting_Point/tips_parse 1-hello.pas | grep -v found
   ```