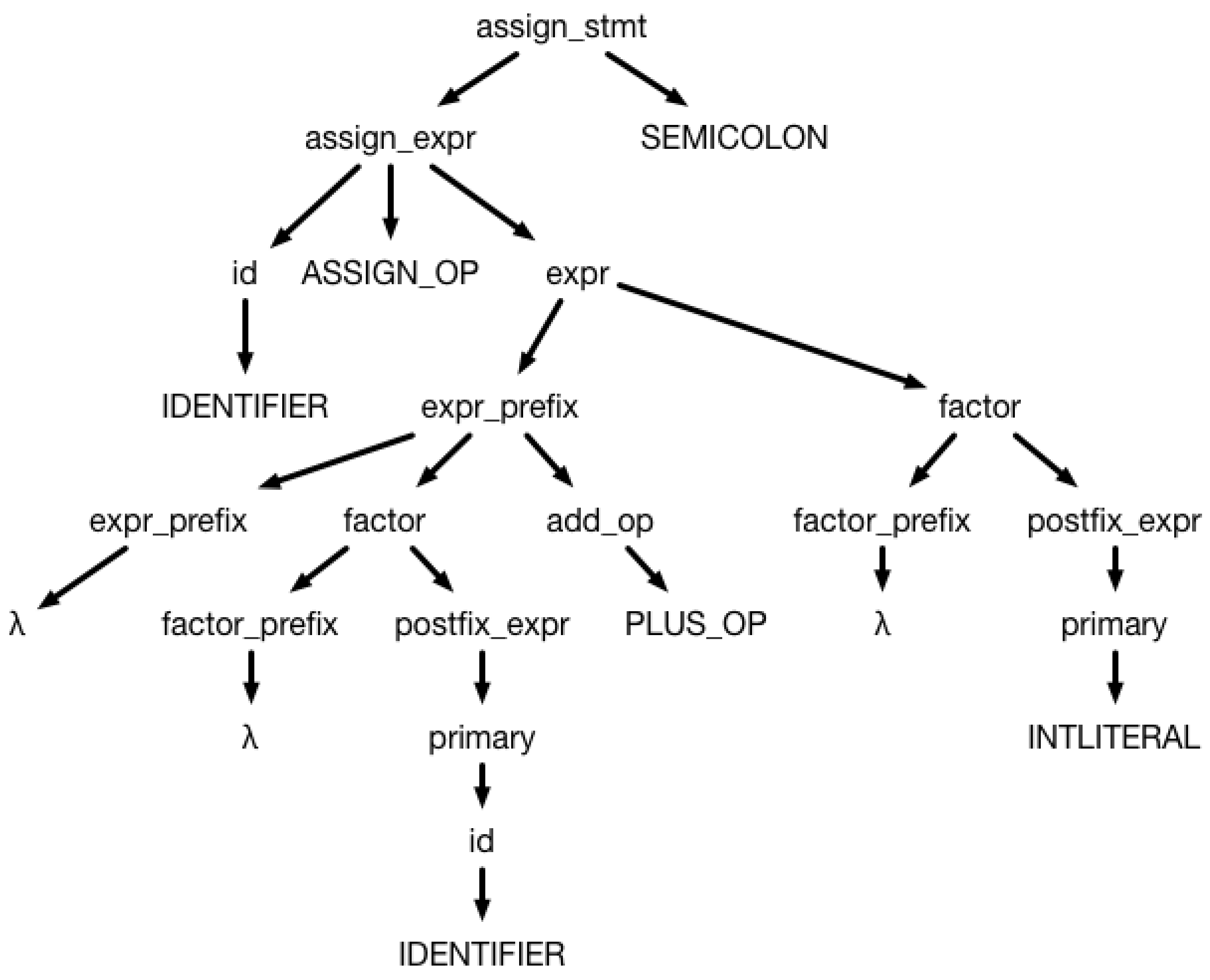
# Project Step 2 — Parser

The second step of the project is building a *parser*. The job of a parser is to convert a stream of tokens (as identified by the scanner) into a *parse tree*: a representation of the structure of the program. So, for example, a parser will convert:

```
A := B + 4;
```

Into a tree that looks something like:



This tree may look confusing, but it fundamentally captures the structure of an *assignment statement*: an assignment statement is an *assignment expression* followed by a semicolon. An assignment expression is decomposed into an *identifier* followed by an assignment operation followed by an *expression*. That expression is decomposed into a bunch of *primary* terms that are combined with addition and subtraction, and those primary terms are decomposed into a bunch of *factors* that are combined with multiplication and division (this weird decomposition of expressions captures the necessary order of operations). Eventually, those *factors* become identifiers or constants.

One important thing to note is that the *leaves* of the tree are the tokens of the program. If you read the leaves of the tree left to right (ignoring lambdas, since they just represent the empty strings), you get:

```
IDENTIFIER ASSIGN_OP IDENTIFIER PLUS_OP INTLITERAL SEMICOLON
```

Which is exactly the tokenization of the input program!

## Context-free grammars

To figure out how each construct in a program (an expression, an if statement, etc.) is decomposed into smaller pieces and, ultimately, tokens, we use a set of rules called a *context-free grammar*. These rules tell us how constructs (which we call "non-terminals") can be decomposed and written in terms of other constructs and tokens (which we call "terminals").

The context-free grammar that defines the structure of LITTLE (the language we are building a compiler for) is:

```
CAPS : CAPS is a token (terminal) made up of one or more characters.
small case symbols are non-terminals.

/* Program */
program           -> PROGRAM id BEGIN pgm_body END
id                -> IDENTIFIER
pgm_body          -> decl func_declarations
decl                   -> string_decl decl | var_decl decl | empty

/* Global String Declaration */
string_decl       -> STRING id := str ;
str               -> STRINGLITERAL

/* Variable Declaration */
var_decl          -> var_type id_list ;
var_type               -> FLOAT | INT
any_type          -> var_type | VOID
id_list           -> id id_tail
id_tail           -> , id id_tail | empty

/* Function Paramater List */
param_decl_list   -> param_decl param_decl_tail | empty
param_decl        -> var_type id
param_decl_tail   -> , param_decl param_decl_tail | empty

/* Function Declarations */
func_declarations -> func_decl func_declarations | empty
func_decl         -> FUNCTION any_type id (param_decl_list) BEGIN func_body
END
func_body         -> decl stmt_list

/* Statement List */
stmt_list         -> stmt stmt_list | empty
stmt              -> base_stmt | if_stmt | while_stmt
base_stmt         -> assign_stmt | read_stmt | write_stmt | return_stmt
```

```
/* Basic Statements */
assign_stmt       -> assign_expr ;
assign_expr       -> id := expr
read_stmt         -> READ ( id_list );
write_stmt        -> WRITE ( id_list );
return_stmt       -> RETURN expr ;

/* Expressions */
expr              -> expr_prefix factor
expr_prefix       -> expr_prefix factor addop | empty
factor            -> factor_prefix postfix_expr
factor_prefix     -> factor_prefix postfix_expr mulop | empty
postfix_expr      -> primary | call_expr
call_expr         -> id ( expr_list )
expr_list         -> expr expr_list_tail | empty
expr_list_tail    -> , expr expr_list_tail | empty
primary           -> ( expr ) | id | INTLITERAL | FLOATLITERAL
addop             -> + | -
mulop             -> * | /

/* Complex Statements and Condition */
if_stmt           -> IF ( cond ) decl stmt_list else_part ENDIF
else_part         -> ELSE decl stmt_list | empty
cond              -> expr compop expr
compop            -> < | > | = | != | <= | >=

/* While statements */
while_stmt       -> WHILE ( cond ) decl stmt_list ENDWHILE
```

So this grammar tells us, for example, that an `if_stmt` looks like the keyword `IF` followed by an open parenthesis, followed by a `cond` expression followed by some `decl` (declarations) followed by a `stmt_list` followed by an `else_part` followed by the keyword `ENDIF`.

An input program matches the grammar (we say "is accepted by" the grammar) if you can use the rules of the grammar (starting from `program`) to generate the set of tokens that are in the input file. If there is no way to use the rules to generate the input file, then the program does not match the grammar, and hence is not a syntactically valid program.

## Building a Parser

There are many tools that make it relatively easy to build a parser for a context free grammar (in class, we will talk about how these tools work): all you need to do is provide the context-free grammar and some actions to take when various constructs are recognized. The tool we are recomending is ANTLR. You should define your grammar in the same `.g4` file in which you defined your lexer.

1. Executing that `.g4` file will produce both a Lexer class and a Parser class.
2. In your `main` file, rather than initializing a lexer and then grabbing tokens from it (as you may have done in step 1), you instead initialize a lexer, initialize a `CommonTokenStream` from that lexer, then initialize a parser *with the CommonTokenStream you just created*.

3. You can then call a function with the same name as your top-level construct (probably `program`) on that parser to parse your input.

## What you need to do

The grammar for LITTLE is given above. All you need to do is have your parser parse the given input file and print `Accepted` if the input file correctly matches the grammar, and `Not Accepted` if it doesn't (i.e., the input file cannot be produced using the grammar rules).