LAB 9 – Recursive Decent Parser

John Dempsey
COMP-232 Programming Languages
California State University, Channel Islands
March 26, 2025
Hard Due Date: April 2, 2025

Read everything before doing anything. In particular, read the submission checklist before you start editing any of the provided code.

You cannot copy scan.c found in LAB 9 to satisfy LAB 8 Scanner assignment. You can study scan.c, learn from it, but you need to write your own scanner program for LAB 8.

Consider the following grammar for a simple programming language:

```
<program> ::= <statement> <EOF> | <statement> <program>
<statement> ::= <assignStmt> | <repeatStmt> | <printStmt>

<assignStmt> ::= <id> = <expr> ;
<repeatStmt> ::= repeat ( <expr> ) <statement>
<printStmt> ::= print ( <expr> ) ;

<expr> ::= <term> | <term> <addop> <expr>
<term> ::= <factor> | <factor> <multop> <term>
<factor> ::= <id> | <number> | <addop> <factor> | ( <expr> )

<number> ::= <int> | <float>
<int> ::= <digit> | <int> <digit>
<float> ::= <digit>. | <digit> <float> | <float> <digit>

<id> ::= <letter> | <id> <letter> | <id> <digit>

<addop> ::= + | -
<multop> ::= * | / | %

<digit> ::= 0-9
<letter> ::= a-z | A-Z | _ | $
```

Refer `input.txt` in the lab download at /home/LAB9 on comp232.com for a sample of statements from the grammar above.

Next, look through `scan.h`; the scanner implemented in `scan.h` and `scan.c` is complete, but you will need to use the tokens it returns. Read the `TOKEN` struct and its referenced `TOKEN_VALUE` union and `TOKEN_TYPE` enum; make sure you understand which `TOKEN_TYPES` will populate which `TOKEN_VALUE` members; if you're uncertain don't hesitate to ask!

Note the addition of an `ungottenToken` to the scanner, and corresponding `ungetToken` function. These serve the same purpose in parsing that `ungetc` served in lexing: when a token is lexed by the scanner and it is determined to not be part of the production being performed it should be "put back" with `ungetToken`, so it can be used as part of the next production.

For instance, consider the production:

```
    <expr> ::= <term> | <term> <addop> <expr>
```

When an expression is parsed, it starts with a term. That term might be followed by an addition-tier operation and then a (smaller) expression, but it might not be! The way to find out is to first parse the leading term, and then check: is the next token is an addition operation? If it is, then the second option in the production above should be carried out, but if it is not an addition operation then the expression being produced is comprised entirely of the leading term, and the token read after that term is not part of the expression. In this case, `ungetToken` should be called on the extra token (which is likely closing parenthesis, a multiplication operation, or a semicolon).

Once you are sure you understand how the statements in `input.txt` fit into the grammar above, how the `TOKEN` struct is set up, and what purpose `ungetToken` serves, you're ready to move on.

Before starting task 1, note that there are multiple enums with similarly-named elements. Be careful not to confuse `TOKEN_TYPES` and `NODE_TYPES`. You will be making nodes in an abstract syntax tree and assigning them types from the `TOKEN_TYPE` enum which can lead to issues. For example, the `ASSIGNMENT_TOKEN` and the `ASSIGN_STMT_NODE` elements are very different. Assigning a node which should be an `ASSIGN_STMT_NODE` the similarly named `ASSIGNMENT_TOKEN` type will lead to errors.

TASK 1 - Parse

Next, open `parse.h`. This header file contains definitions for a `NUMBER` struct and a `NODE` struct.

The `NUMBER` struct is to store numerical values, which in the grammar above are either `ints` or `floats` (stored in `long` and `double` form respectively in the `NUMBER` struct).

The `NODE` struct defines an individual node in an abstract syntax tree. The goal of parsing is to use the token's output by the scanner to build an abstract syntax tree which adheres to the grammar.

You will be implementing a **recursive descent parser**. The syntax tree set up by this particular recursive descent parser will be a binary tree; each node will have at most 2 children (named `leftChild` and `rightChild`). For example, if a node is of type `ASSIGN_STMT_NODE`, its `leftChild` will be the `IDENT_NODE` representing the variable being assigned value and its `rightChild` will be the `EXPR_NODE` representing the expression whose value is to be stored in the variable.

`parse.h` also includes these function declarations:

```
        NODE *program();
        NODE *statement();
        NODE *assignStmt(TOKEN **currToken);
        NODE *repeatStmt(TOKEN **currToken);
        NODE *printStmt(TOKEN **currToken);
        NODE *expr(TOKEN **currToken);
        NODE *term(TOKEN **currToken);
        NODE *factor(TOKEN **currToken);
        NODE *ident(TOKEN **currToken);
        NODE *number(TOKEN **currToken);
```

The names of these functions coincide with the productions (above the token level) in the grammar. This is not a coincidence. Each function above will perform the corresponding production. Your first task is to complete the definitions of these functions in `parse.c`. The `program` and `statement` functions have already been completed as examples.

Before you start, note the functions `freeToken` and `getNextToken`. As you'd expect `freeToken` frees the space allocated by the input token. `getNextToken` takes as an argument the previous token (which is freed) and then returns the next token. Every token needs to be freed at some point; if a token is ever passed into a `getNextToken` call, that will free it, so `freeToken` does not need to be called on it again.

Similarly, if a token is passed into one of the production functions above, that production will "consume" (i.e. parse and free) the token. If, however, a token is never passed into a `getNextToken` call or a production function, then it should be freed manually with `freeToken`. This will happen with trailing tokens, such as semicolons, right parenthesis, etc.

It is often necessary to use the leading token to decide which productions to call. See the completed `statement` function; if the first token is an `IDENT_TOKEN` then the statement being produced must be an assignment statement, as this is the only type of statement which starts with an `IDENT_TOKEN`. Similarly, if the first token is a `REPEAT_TOKEN` then the statement must be a repeat statement, and so on.

There are tokens which do not need to be stored in the syntax tree, but **they are still necessary for valid parsing**. For instance, the `ASSIGNMENT_TOKEN`, which is the second token in an assignment statement, is not stored in the syntax tree.

Your parser must still get the token and ensure that it is an assignment token, and throw an error (see the `error` function in `parse.c`) if it is not the correct type of token. This will also often be the case for parenthesis and semicolons. To be clear, these tokens are not redundant or useless; their purpose is syntactic, and without them parsing could not be done without ambiguity, but once they've been used for parsing their purpose has been fulfilled and their usefulness exhausted.

Productions (other than `program` and `statement`, which serve as "entry points") should have the first token in the production passed in as an argument, so they should not have to call `getNextToken` to get the first token in the production. Similarly, when a production is complete, the last token that it used should be freed unless it is passed into another production, as discussed above. In other words, if the token gotten from the last call to `getNextToken` in a given production is not passed into another production, it should be freed using `freeToken`.

You may use the gdb debugger to check your `parse.c` before continuing. However, you may find it easier to instead complete the next task (which prints the parse tree) and then debug the two together.

Keep in mind that we're building a recursive setup here, so these production functions will be calling each other constantly, but none of the functions should be very complex for this reason. Consider parsing an expression; the expression starts with a term, for sure, so our first step is to parse a term. How do we parse a term? Who cares, we have another function for that, so we can just call that other function to parse the leading term (and trust our past/future selves to fill out that function correctly). If you're creating a node, and you find yourself accessing at the data contained in its children, you're overcomplicating things.

TASK 2 - Print the Parse Tree

Check out `print.h` and `print.c`. These declare and define functions to recursively print the parse tree constructed in the previous task.

They should be somewhat self-explanatory, the structure is nearly identical to that of parsing.

The main difference is the extra `indent` argument. It exists solely for aesthetic purposes; it determines how far the printed node will be indented. To adhere to the sample output below, sub-productions should be indented 1 further than their parent productions, so you'll often find yourself passing `indent+1` into your print functions.

Like the previous task, `printProgram` and `printStatement` have been completed for you. You should follow the example in `printStatement` to decide what to print. Every node other than the program node should print start and end messages like the statement node does.

The contents of `input.txt` are:

```
firstVar = 100;
print(firstVar);
second2var = 0.15;
print(second2var);
repeat (2)
    firstVar = firstVar + 1;
print(+firstVar);
repeat(10)
    print(-101.725);
```

When the `parse_print_tree` executable is run with the above `input.txt` as an argument, it should produce the result below. You may find it useful to go through the input above, statement by statement, and compare to the output below before starting.

```
Done parsing...

START STATEMENT
|  START ASSIGN STATEMENT
|  |  START IDENT
|  |  |  firstVar
|  |  END IDENT
|  |  START EXPRESSION
|  |  |  START TERM
|  |  |  |  START FACTOR
|  |  |  |  |  START NUMBER
|  |  |  |  |  |  INT : 100
|  |  |  |  |  END NUMBER
|  |  |  |  END FACTOR
|  |  |  END TERM
|  |  END EXPRESSION
|  END ASSIGN STATEMENT
END STATEMENT
START STATEMENT
|  START PRINT STATEMENT
|  |  START EXPRESSION
|  |  |  START TERM
|  |  |  |  START FACTOR
|  |  |  |  |  START IDENT
|  |  |  |  |  |  firstVar
|  |  |  |  |  END IDENT
|  |  |  |  END FACTOR
|  |  |  END TERM
|  |  END EXPRESSION
|  END PRINT STATEMENT
END STATEMENT
START STATEMENT
|  START ASSIGN STATEMENT
|  |  START IDENT
|  |  |  second2var
|  |  END IDENT
|  |  START EXPRESSION
|  |  |  START TERM
|  |  |  |  START FACTOR
|  |  |  |  |  START NUMBER
|  |  |  |  |  |  FLOAT : 0.150000
|  |  |  |  |  END NUMBER
|  |  |  |  END FACTOR
|  |  |  END TERM
|  |  END EXPRESSION
|  END ASSIGN STATEMENT
END STATEMENT
START STATEMENT
|  START PRINT STATEMENT
|  |  START EXPRESSION
|  |  |  START TERM
|  |  |  |  START FACTOR
|  |  |  |  |  START IDENT
|  |  |  |  |  |  second2var
|  |  |  |  |  END IDENT
|  |  |  |  END FACTOR
|  |  |  END TERM
|  |  END EXPRESSION
|  END PRINT STATEMENT
END STATEMENT
START STATEMENT
|  START REPEAT STATEMENT
|  |  START EXPRESSION
|  |  |  START TERM
|  |  |  |  START FACTOR
|  |  |  |  |  START NUMBER
|  |  |  |  |  |  INT : 2
|  |  |  |  |  END NUMBER
|  |  |  |  END FACTOR
|  |  |  END TERM
|  |  END EXPRESSION
|  |  START STATEMENT
|  |  |  START ASSIGN STATEMENT
|  |  |  |  START IDENT
|  |  |  |  |  firstVar
|  |  |  |  END IDENT
|  |  |  |  START EXPRESSION
|  |  |  |  |  START TERM
|  |  |  |  |  |  START FACTOR
|  |  |  |  |  |  |  START IDENT
|  |  |  |  |  |  |  |  firstVar
|  |  |  |  |  |  |  END IDENT
|  |  |  |  |  |  END FACTOR
|  |  |  |  |  END TERM
|  |  |  |  <ADDOP +>
|  |  |  |  |  START EXPRESSION
|  |  |  |  |  |  START TERM
|  |  |  |  |  |  |  START FACTOR
|  |  |  |  |  |  |  |  START NUMBER
|  |  |  |  |  |  |  |  |  INT : 1
|  |  |  |  |  |  |  |  END NUMBER
|  |  |  |  |  |  |  END FACTOR
|  |  |  |  |  |  END TERM
|  |  |  |  |  END EXPRESSION
|  |  |  |  END EXPRESSION
|  |  |  END ASSIGN STATEMENT
|  |  END STATEMENT
|  END REPEAT STATEMENT
END STATEMENT
START STATEMENT
|  START PRINT STATEMENT
|  |  START EXPRESSION
|  |  |  START TERM
|  |  |  |  START FACTOR
|  |  |  |  |  <ADDOP +>
|  |  |  |  |  START FACTOR
|  |  |  |  |  |  START IDENT
|  |  |  |  |  |  |  firstVar
|  |  |  |  |  |  END IDENT
|  |  |  |  |  END FACTOR
|  |  |  |  END FACTOR
|  |  |  END TERM
|  |  END EXPRESSION
|  END PRINT STATEMENT
END STATEMENT
START STATEMENT
|  START REPEAT STATEMENT
|  |  START EXPRESSION
|  |  |  START TERM
|  |  |  |  START FACTOR
|  |  |  |  |  START NUMBER
|  |  |  |  |  |  INT : 10
|  |  |  |  |  END NUMBER
|  |  |  |  END FACTOR
|  |  |  END TERM
|  |  END EXPRESSION
|  |  START STATEMENT
|  |  |  START PRINT STATEMENT
|  |  |  |  START EXPRESSION
|  |  |  |  |  START TERM
|  |  |  |  |  |  START FACTOR
|  |  |  |  |  |  |  <ADDOP ->
|  |  |  |  |  |  |  START FACTOR
|  |  |  |  |  |  |  |  START NUMBER
|  |  |  |  |  |  |  |  |  FLOAT : 101.725000
|  |  |  |  |  |  |  |  END NUMBER
|  |  |  |  |  |  |  END FACTOR
|  |  |  |  |  |  END FACTOR
|  |  |  |  |  END TERM
|  |  |  |  END EXPRESSION
|  |  |  END PRINT STATEMENT
|  |  END STATEMENT
|  END REPEAT STATEMENT
END STATEMENT

Process finished with exit code 0
```

The vertical bars marking indentation levels are handled by the `printfIndent` function defined at the bottom of `print.c`. If you feel like going down a rabbit hole completely unrelated to the lab, explore how the `printfIndented` function works to use varargs in a format string like `printf` does.

Much like the previous task, we are building a recursive set of functions, and that should keep the individual functions somewhat simple; if you find yourself accessing a node's childrens' data in a print function, you should probably be calling one of the other print functions instead.

When you've defined every function in `print.c`, edit the run configurations for the `parse_print_tree` executable to use a path from within the `cmake-build-debug` folder to `input.txt` as the first argument, and then run it!

You should fully debug tasks 1 and 2 before moving on to task 3.

TASK 3 - Evaluate

Finally, we can evaluate the completed parse tree to perform the actions specified in `input.txt`.

Open `eval.h` and `eval.c`. In `eval.h`, see the following function declarations:

```
void evalProgram(NODE *node);
void evalStatement(NODE *node);
void evalAssignStmt(NODE *node);
void evalRepeatStmt(NODE *node);
void evalPrintStmt(NODE *node);
NUMBER evalExpr(NODE *node);
NUMBER evalTerm(NODE *node);
NUMBER evalFactor(NODE *node);
NUMBER evalNumber(NODE *node);
NUMBER evalId(NODE *node);
NUMBER evalOperation(NUMBER operand1, NUMBER operand2, char op);
```

These functions will coincide with those implemented in `parse.c` and `print.c`; they will traverse the parse tree recursively. This time, the goal is neither to build the tree nor to print it, but to **perform the actions specified by the parsed statements**. That is, in this task we are actually executing the commands in `input.txt`. When this task is complete we will have a fully functional interpreter! Just don't try to give it any tokens with more than 128 characters 😉.

The intended functionality of assignment, print, and repeat statements is likely intuitive, but we will define them anyway:

- `<ident> = <expr> ;`:
    - Assign the value of `<expr>` to a symbol (i.e. variable) with identifier `<ident>`.
        - If the variable is already defined, its value should be changed.
        - Otherwise, a new variable should be created and assigned the value.
    - Check out the `SYMBOL_TABLE_NODE` struct in `eval.h`; it stores variables (i.e. identifiers with values) in a linked list.
    - Also check out these completed functions at the bottom of `eval.c` (you'll need to use them to manage variables)
        - `findSymbol`
        - `createSymbol`
        - `reassignSymbol`
        - `addSymbolToTable`
        - `evalSymbol`

- `repeat ( <expr> ) <statement>`:

    - If the `<expr>` evaluates to an `INT_TYPE`, perform the action(s) specified in `<statement>` that many times (i.e. eval `<statement>` that many times, in a loop).
    - If the `<expr>` evaluates to a `FLOAT_TYPE`, treat it as an integer (ignore the fractional part).
    - If the evaluated `<expr>` is negative, it should be treated as if it is zero.

- `print ( <expr> ) ;`:
    - Print the value of the evaluated `<expr>` in the console.
    - Should include both the `NUMBER_TYPE` and the corresponding value of the number.
    - See the sample run below for a sample format.

The remaining functions are for the evaluation of expressions and their contained terms, factors, numbers, and variables. Pay attention to the types specified for the functions; `evalExpr` is of type `NUMBER` because the result of an evaluated expression is a `NUMBER`. `evalAssignStmt`, on the other hand, is of type `void` because the function of an assignment statement is to **do something** (specifically, assign the value of an expression to a variable) and **not to return something**.

Note that `evalId` is for accessing the value of a variable when an ident node is the child of a factor. `evalId` should not be used when an ident node is the child of an assignment node, as in this case the goal is to assign a value to the variable, not find a previously assigned value. If `evalId` is called and there is not variable with the specified identifier, return the `NAN_NUMBER`. `NAN_NUMBER` is defined at the top of `eval.c`, and is an instance of the `NUMBER` struct of type `FLOAT_TYPE` with value `NAN` (short for "not a number").

There is also a set of functions which doesn't have an equivalent in the previous tasks: `evalOperation` and its referenced helper functions. This function takes in two `NUMBER`s and a character `+ - * / %` representing and operation, and simply calls the corresponding evaluation function `evalAdd`, `evalSub`, `evalMult`, `evalDiv`, or `evalMod`. You will need to fill out these evaluation functions.

If both of an operation's inputs are numbers of type `INT_TYPE`, then the result should be an `INT_TYPE`. If, on the other hand, either of the operands is `FLOAT_TYPE` then the output should be a `FLOAT_TYPE` (as is the convention in most languages). When the modulus `%` operation is used with floats, it should return the result of the `fmod` function from `math.h` (check out the manual entry). When division is done with integers, it should perform floor division.

Once again: recursive setup --> somewhat simple functions. If you're looking at childrens' data instead of just passing the children into a different function, you're working too hard.

When you're ready to run, edit the configurations for the `parse_eval` executable to include a path to `input.txt` as an argument, and run!

With the provided contents of `input.txt`, the output should be:

```
Done parsing...

INT : 100
FLOAT : 0.150000
INT : 102
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000
FLOAT : -101.725000

Process finished with exit code 0
```

Submission Checklist

In your submission:

- sftp all source code into /home/<your user name>/LAB9 directory on comp232.com.
- Nothing should be edited outside of functions that had `TODO`s in them.
    - This refers to the provided / completed functions; they should not be edited.
    - You are certainly welcome to add functions to `parse.c`, `print.c` and `eval.c`, but not to the `.h` files.
    - Do not add global variables, and do not edit `print_test.c` or `eval_test.c`.
- `parse.c` should be completed.
- `print.c` should be completed.
- When `parse_print_tree` is run with the provided `input.txt`, it should:
    - run without error (exit code 0).
    - match the provided sample run.

    - run without memory issues.
- A screenshot of your `parse_print_tree` run with the provided `input.txt` should be included.
- `eval.c` should be completed.
- When `parse_eval` is run with the provided `input.txt`, it should:
    - run without error (exit code 0).
    - match the provided sample run.
- A screenshot of your `parse_eval` run with the provided `input.txt` should be included.
- In a text file, provide the answer the following:
    - In parsing expressions, we've gone through the contained terms from left to right; with consequently the right-most terms are at the bottom of the tree. Similarly, in parsing terms, we've gone through the contained factors from left from left to right, so the right-most factors are at the bottom of the tree. What issues does this create in evaluation? Provide an example of an expression which will be evaluated incorrectly (arithmetically speaking) by your completed parsing and evaluation utilites. What should it evaluate to (arithmetically), what does it evaluate to instead (correctly with respect to the grammar, but incorrectly with respect to arithmetics), and why?

Disclaimer on "matching the sample runs": I change these labs every semester; if I've made an obvious mistake like, say, changing an identifier or a numeric value in the input without updating the sample run, then please inform me and don't worry about matching my errors. I believe I've found them all, but I always believe that and it's never true.

input2.txt

```
firstvar = 1;

secondvar = 2;

repeat (10)

 thirdvar = 2 * (firstvar % secondvar) / (firstvar + 2);;

repeat (firstvar + 2 * secondvar)

 repeat (thirdvar)

  print -firstvar;;;
```

sample_output.txt

```
Done parsing...
```

```
=> START program
=> START statement
=> START assignment
=> START identifier
<id> firstvar
=> END identifier
=> START expression
=> START term
=> START factor
=> START number
<number> 1.000000
=> END number
=> END factor
=> END term
=> END expression
=> END assignment
=> END statement
=> START program
=> START statement
=> START assignment
=> START identifier
<id> secondvar
=> END identifier
=> START expression
=> START term
=> START factor
=> START number
<number> 2.000000
=> END number
=> END factor
=> END term
=> END expression
=> END assignment
=> END statement
=> START program
=> START statement
=> START repeat
=> START expression
=> START term
=> START factor
=> START number
<number> 10.000000
=> END number
=> END factor
=> END term
=> END expression
=> START statement
=> START assignment
=> START identifier
<id> thirdvar
=> END identifier
=> START expression
=> START term
=> START factor
=> START number
<number> 2.000000
=> END number
=> END factor
<times>
=> START expression
=> START term
=> START factor
=> START expression
=> START term
=> START factor
=> START identifier
<id> firstvar
=> END identifier
=> END factor
<modulus>
=> START expression
=> START term
=> START factor
=> START identifier
<id> secondvar
=> END identifier
=> END factor
=> END term
=> END expression
=> END term
=> END expression
=> END factor
<divide>
=> START expression
=> START term
=> START factor
=> START expression
=> START term
=> START factor
=> START identifier
<id> firstvar
=> END identifier
=> END factor
=> END term
<plus>
=> START expression
=> START term
=> START factor
=> START number
<number> 2.000000
=> END number
=> END factor
=> END term
=> END expression
=> END expression
=> END factor
=> END term
=> END expression
=> END term
=> END expression
=> END term
=> END expression
=> END assignment
=> END statement
=> END repeat
=> END statement
=> START program
=> START statement
=> START repeat
=> START expression
=> START term
=> START factor
=> START identifier
<id> firstvar
=> END identifier
=> END factor
=> END term
<plus>
=> START expression
=> START term
=> START factor
=> START number
```

```
<number> 2.000000
=> END number
=> END factor
<times>
=> START expression
=> START term
=> START factor
=> START identifier
<id> secondvar
=> END identifier
=> END factor
=> END term
=> END expression
=> END term
=> END expression
=> END expression
=> START statement
=> START repeat
=> START expression
=> START term
=> START factor
=> START identifier
<id> thirdvar
=> END identifier
=> END factor
=> END term
=> END expression
=> START statement
=> START print
=> START expression
=> START term
=> START factor
<minus>
=> START factor
=> START identifier
<id> firstvar
=> END identifier
=> END factor
=> END factor
=> END term
=> END expression
=> END print
=> END statement
=> END repeat
=> END statement
=> END repeat
=> END statement
=> START program
=> END program
=> END program
=> END program
=> END program
=> END program
```

The following files can be downloaded at comp232.com in the /home/LAB9 directory.

```
root@comp232:/home/LAB9# ls -l
total 48
-rw-r--r-- 1 john john 3661 Mar 30 10:27 eval.c
-rw-r--r-- 1 john john  926 Mar 30 10:27 eval.h
-rw-r--r-- 1 john john  261 Mar 30 10:27 eval_test.c
-rw-r--r-- 1 john john  166 Mar 30 10:27 input.txt
-rw-r--r-- 1 john john 2495 Mar 30 10:27 parse.c
-rw-r--r-- 1 john john 1146 Mar 30 10:27 parse.h
-rw-r--r-- 1 john john 1814 Mar 30 10:27 print.c
-rw-r--r-- 1 john john  563 Mar 30 10:27 print.h
-rw-r--r-- 1 john john  240 Mar 30 10:27 print_test.c
-rw-r--r-- 1 john john 5549 Mar 30 10:27 scan.c
-rw-r--r-- 1 john john  839 Mar 30 10:27 scan.h
```