To goal of this project is to implement in C or C++ (your choice) an interpreter that evaluates arithmetic expressions with variables in local scopes. The local scope is similar to the `let` construct in functional languages:

```
let x = 2 in
  x * x
end
```

which evaluates to 4,

```
3 * ( let x = 2 in
    x * x
  end )
```

which evaluates to 12, and

```
3 * ( let x = 2 in
    ( let y = x * x + 1 in
      y / 2
    end )
  end )
```
evaluates to 9 (integer division rounds towards zero).

**Expression grammar**

Consider our familiar augmented LL(1) grammar for an expression language (see “Syntax” lecture notes on the LL(1) expression grammar):

```
<table>
<thead>
<tr>
<th>Grammar Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;expr&gt; -&gt; &lt;term&gt; &lt;term_tail&gt;</td>
<td>term_tail.subtotal := term.value; expr.value := term_tail.value</td>
</tr>
<tr>
<td>&lt;term&gt; -&gt; &lt;factor&gt; &lt;factor_tail&gt;</td>
<td>factor_tail.subtotal := factor.value; term.value := factor_tail.value</td>
</tr>
<tr>
<td>&lt;term_tail1&gt; -&gt; '+' &lt;term&gt; &lt;term_tail2&gt;</td>
<td>term_tail2.subtotal := term_tail1.subtotal + term.value; term_tail1.value := term_tail2.value</td>
</tr>
<tr>
<td></td>
<td>'-' &lt;term&gt; &lt;term_tail2&gt;</td>
</tr>
<tr>
<td></td>
<td>empty</td>
</tr>
<tr>
<td>&lt;factor1&gt; -&gt; '(' &lt;expr&gt; ')'</td>
<td>factor1.value := expr.value</td>
</tr>
<tr>
<td></td>
<td>'-' &lt;factor2&gt;</td>
</tr>
<tr>
<td></td>
<td>num</td>
</tr>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td>&lt;factor_tail1&gt; -&gt; '*' &lt;factor&gt; &lt;factor_tail2&gt;</td>
<td>factor_tail2.subtotal := 1</td>
</tr>
</tbody>
</table>
```
factor_tail1.subtotal*factor.value;
factor_tail1.value := factor_tail2.value

| '/' <factor> <factor_tail2> | factor_tail2.subtotal :=
| | factor_tail1.subtotal/factor.value;
factor_tail1.value := factor_tail2.value

| empty | factor_tail1.value := factor_tail1.subtotal

where lookup(id.name) returns the value of the identifier (assuming it is a variable with a value) from a name-value store such as a “binding stack”.

Note that in the augmented grammar above we use indexed nonterminals which is needed to distinguish the use of the nonterminals in the semantic rules, so factor1 and factor2 are just the same factor nonterminal. The meta-symbol empty in the productions denotes $\varepsilon$.

Let-rules

To facilitate scoping of variables with let, we need to store the name-value pairs on a “binding stack”. When our interpreter executes a let the variable declared is only visible in the body of the let expression. Therefore, the obvious choice for a name-value store would be a binding stack, where a new name-value pair is pushed on the stack when the interpreter enters the let and pops it off when it leaves the let. When the variable is used in the body of the let the interpreter searches the stack from the top (the last pair added) to the bottom to find the variable. This also allows for “shadowing” in scope rules when scopes are nested:

```latex
let x = 1 in
  ( let x = 3 in
    2*x
  end )
+ x
end
```

which evaluates to 7. Here the scope of the $x$ in the outer let is hidden in the inner body of the inner let.

When the interpreter needs the value of a variable in an expression it looks it up in the binding stack from the top to the bottom to find the most recently pushed variable.

To add the let construct to our grammar we extend it with a new production:

<table>
<thead>
<tr>
<th>GRAMMAR PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;expr1&gt; -&gt; let id = &lt;expr2&gt; in &lt;push&gt; &lt;expr3&gt; end &lt;pop&gt;</td>
<td>push.name = id.name;</td>
</tr>
<tr>
<td></td>
<td>push.value = expr2.value;</td>
</tr>
<tr>
<td></td>
<td>expr1.value = expr3.value</td>
</tr>
</tbody>
</table>

We use the “dummy” nonterminals <push> and <pop> that represent the push and pop operations that we need to execute before and after the evaluation of the let body expression expr3:
Note that in this way the interpreter performs the push and pop operations cleverly while parsing the input, as these are embedded in the augmented grammar.

More specifically, the stack object we use is a global stack that is updated with push and pop operations and can be searched top-down with a lookup function. You have to decide on a stack implementation. The stack size should be at least 100.

Recursive descent parsing

In this project we will use the recursive-descent parsing technique. The recursive descent parser for the let construct uses a function for each nonterminal to parse the input that the nonterminal represents, for example parsing an <expr>:

```c
/* define keyword tokens recognized by the scanner */
#define LET 256
#define IN  257
#define END 258

/* the <expr> parser */
int expr()
{
  int result;
  if (lookahead() == LET)
  {
    char *name;
    int value;
    match(LET);  // let
    name = id();  // id
    match('=');   // =
    value = expr();  // <expr>
    match(IN);    // in
    push(name, value); // <push>
    result = expr();  // <expr>
    match(END);   // end
    pop();       // <pop>
  }
  else
  {
    int value = term();  // <term>
    result = term_tail(value);  // <term_tail>
  }
  return result;
}
```

You need to complete the recursive descent parser implementation for the augmented grammar. For this assignment you can write the code in C or C++.

The `match()` function can be written based on these:

```c
void match(int token)
{
  if (lookahead() == token)
    getnext();
  else
    ... // report syntax error and exit
}
Adding a scanner

You need to write a scanner first to tokenize the input. The scanner has two functions: lookahead() and getnext(). The lookahead() returns an integer that represents the current token. The getnext() moves to the next token on the input.

Note that if all tokens were single ASCII we can simply use getchar() for getnext() to read the next ASCII token:

```c
int current = 0;
// getnext() gets the next token (simple version)
void getnext()
{
    do
        current = getchar();
    while (isspace(current)); // skip space until next char
}
// lookahead() returns current token
int lookahead()
{
    if (current == 0) // first time around
        getnext();
    return current;
}
```

However, we also need to recognize the let, in, end keywords, identifier names, and integer constants. For identifier names (variables), we need to store the current variable name and set current to ID or to NUM for integer constants:

```c
#define ID 259
#define NUM 260
int current = 0;
int number;
char name[80]; // max identifier name length is 79
// getnext() improved version
void getnext()
{
    int c;
    do
        c = getchar();
    while (isspace(c));
    if (...check for keyword...) {
        current = ...keyword token...
    } else if (...check for identifier...) {
        current = ID;
        ... fill name[] with id name
    } else if (...check for number...) {
        current = NUM;
        ... number = decoded value
    } else {
        current = c;
    }
}```
Check for the keywords **let**, **in**, and **end** to set **current** to the value of **LET**, **IN**, or **END** (these are macros, see above). Note that **EOF** is returned by **lookahead()** upon end of file, which will be used to stop the expression parser and print the calculated value.

You should test your scanner implementation first by calling **getnext()** in a loop and printing the **lookahead()** token:

```c
while (lookahead() != EOF)
{
    switch (lookahead())
    {
        case ID:
            printf("ID=%s
", name);
            break;
        case NUM:
            printf("NUM=%d
", number);
            break;
        default:
            printf("%c
", lookahead());
    }
    getnext();
}
```

This should tokenize the stdin stream and print the results to stdout. When you are confident that the scanner works properly, you can start working on the next phase, the interpreter to evaluate expressions.

**Interpreter**

Implement the let-expression interpreter using your scanner and parser. The parser should use recursive descent. Implement a stack to push and pop name-value pairs and search the stack top-down for the value of a name. The input to the interpreter is read from the stdin. The value should be printed to stdout.

**BONUS**

You can earn 10% extra credit as follows. Modify the grammar to include multiple variables in a **let** such as **let x = 1, y = 2 in x+y end** by starting with:

```plaintext
<table>
<thead>
<tr>
<th>GRAMMAR PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;expr1&gt; -&gt; let &lt;vars&gt; in &lt;expr2&gt; end &lt;popvars&gt;</td>
<td>expr1.value = expr2.value</td>
</tr>
<tr>
<td>&lt;vars&gt; -&gt; id '=' &lt;expr&gt; &lt;restvars&gt;</td>
<td>vars.count = restvars.count + 1</td>
</tr>
<tr>
<td>&lt;restvars&gt; -&gt; , &lt;vars&gt;</td>
<td>restvars.count = vars.count</td>
</tr>
<tr>
<td></td>
<td>restvars.count = 0</td>
</tr>
<tr>
<td></td>
<td>???</td>
</tr>
<tr>
<td>&lt;popvars&gt; -&gt; empty</td>
<td>???</td>
</tr>
</tbody>
</table>
```

The semantic rules are not complete, you will need to fill in the code for `???`. The number of pops to execute in `<popvars>` should be the number of pushed bindings that `<vars>` executed. Use the `count` attribute for this as suggested above. Implement the new grammar and rules in your recursive descent parser.

– End