Late homework assignments will not be accepted, unless you have a valid written excuse (medical, etc.). You must do this assignment alone. No team work or "talking with your friends" will be accepted. No copying from the Internet. Cheating means zero.

==========

SUBMITTING THE HOMEWORK ASSIGNMENT

For this assignment you have to create a single file called parser.c. This file must be in a directory that you name using your student ID number. This directory must therefore contain the file parser.c and nothing else.

Once you are done answering the assignment then create an archive or zip file of the directory with the single file parser.c in it, just like you did in homework assignment 1, and upload the archive on iSpace. Again the archive or zip file must be named using your student ID number. If you do not remember how to do that then read the instructions for homework assignment 1 again.

Following this naming convention is very important, as I will use a computer program to automatically compile, run, and test your parser (on many different input programs), and print your homework assignment. Submitting files with the wrong names will result in your assignment not being processed correctly by my program, which means you will get a zero and you will have submitted your homework assignment for nothing. I will not have time to change the program I use just because one or two students failed to follow the instructions in this homework assignment. It is YOUR responsibility to ensure that the files you submit are the right files with the right names.

==========

HOMEWORK ASSIGNMENT - A top-down (recursive) predictive parser.

Consider the following context-free grammar for a small programming language:

```
    B -> { L }

    L -> D ; L
    L -> S ; L
    L -> eps

    D -> INT ID = E

    S -> IF ( E ) B ELSE B
    S -> ID = E

    E -> T E'

    E' -> + T E'
    E' -> eps

    T -> F T'

    T' -> * F T'
    T' -> eps

    F -> ( E )
    F -> ID
    F -> NUM
```

where {, }, ;, INT, ID, =, IF, (,), ELSE, +, *, and NUM are tokens, and eps represents the empty string epsilon. B is the start symbol of the grammar and represents a block of code, between curly braces. A block of code is then a list L of definitions and statements, in any order. A definition D is simply the definition of an integer variable which is initialized using an arithmetic expression E. A statement S is either an IF statement (with one block of code for the "then" part and one block of code for the "else" part) or the assignment of an arithmetic expression E to a variable. Arithmetic expressions follow the same grammar that we have

used in class (except for the extra production F -> NUM, where NUM is a token representing a natural number).

This grammar is left-factored and does not have any left recursion. In fact you will see that the parser you are going to write for this grammar can always decide which grammar production to use next during every step of the parsing process by always using at most one token of lookahead, which shows that the grammar above is LL(1). This in turn means that you can write a top-down (recursive) predictive parser for it (a top-down recursive descent parser that does not require any backtracking). You then know that the shape of the C code that you have to write for this predictive parser will exactly follow the shape of the LL(1) grammar above.

To do this, create a file named parser.c. This file must start with the following code:

```
#include "tokens.h"
#include "tree.h"
#include "varim.h"
#include "parser.h"
```

The header file tokens.h defines the integer macros INT, ID, IF, ELSE, and NUM that you will use in your code to represent the corresponding tokens. All tokens in your code must be represented using either single characters constants (see below), or integer macros like INT or NUM that are defined in the file tokens.h.

The file lexer.l specifies the lexer that you have to use in your program. For multi-characters lexemes (such as 273 or abcd) the lexer returns one of the integer token values defined in tokens.h (such as NUM or ID). For single-character lexemes (such as ; or +) the lexer uses the C code *yytext (which is the same as doing yytext[0] in C) to use the content of the first element of the variable yytext as the token. Since we know that yytext contains the original lexeme from the source program, what the code in lexer.l does in fact is to use the integer ASCII code of the lexeme to represent the token. This is a common trick in C.

For example, when the lexer sees in the input the character +, it matches it with the regular expression [{};()=+*]. The character + is then copied into the first element of yytext and the corresponding action (C code) is executed. The action then simply takes the ASCII code of the character + (i.e. the integer 43) that was just stored in the first element of yytext and returns to the parser that ASCII code as the token (since tokens are represented as integers). It also turns out that in C, you can directly get the ASCII code of a character by using a character constant. So, for example, to get the ASCII code for the character +, you just have to use the constant '+' (the same character, wrapped inside single quotes). You can use that then in your file parser.c instead of having to remember the values of the different ASCII codes for the different single-character lexemes.

In short, if your parser needs to manipulate a multi-character lexeme like 273 or abcd then your parser needs to use the corresponding C macros like NUM or ID that are defined in tokens.h. If your parser needs to manipulate a single-character lexeme like ; or + then you parser needs to use the corresponding ASCII character constants like ';' or '+'. DO NOT USE INTEGERS LIKE 8107 OR 43 DIRECTLY IN YOUR CODE: ONLY USE THE MACROS DEFINED IN tokens.h AND ASCII CHARACTER CONSTANTS.

The file tokens.h also contains the prototypes for three functions that you will need to use in your parser: raise_syntax_error, consume_next_input_token, peek_at_next_input_token. You will use these functions in exactly the same way as we used them in class (or in the lecture notes). The only difference is that the function raise_syntax_error takes here two arguments: when your parser detects a syntax error, it must call this function with, as arguments, the token it expected to see next in the input, and the token it actually found next in the input. The function raise_syntax_error will then be able to give you good error messages, which will help you debug your parser.

The code for these three functions is in the file utils.c, but you do not

have to look at the content of that file to do the homework assignment. Just use the functions provided in tokens.h when your parser needs to peek at the next token in the input, or needs to consume the next token in the input, or needs to raise a syntax error.

The header file tree.h defines a type called ptn. This is the type that you must use in your parser code to represent Parse Tree Nodes: either nodes for tokens at the leaves of the parse tree, or nodes for nonterminals as the interior nodes of the parse tree. A parse tree is then made of many parse tree nodes connected together with pointers.

As it parses the input program (or, more precisely, as it parses the tokens it receives from the lexer) your parser will have to build the parse tree that describes the structure of the input. All the parse trees (and sub-trees of parse trees) that you will create in your parser will use the type ptn. To create the various parse tree nodes for the parse tree, you will use the functions new_token_node, new_epsilon_node, new_B_node, new_L_node, new_D_node, new_S_node, new_E_node, new_Ep_node, new_T_node, new_Tp_node, and new_F_node which are described in the file tree.h.

For example, if your parser needs to construct a parse tree node based on the grammar production:

    E' -> + T E'

then your code will build the three parse trees for the token + and the two nonterminals T and E', and then use the new_Ep_node function with three arguments (the three ptn representing the three subtrees for E') to build the node for E' itself.

Note: the function prototypes in the file tree.h are commented out. Do not worry about this, just use the functions as if they were not commented out, and it will work fine. I had to do it this way to correctly simulate function overloading in C, since the functions like new_L_node or new_F_node may take different number of arguments at different times.

The header file tree.h also describes the function print_parse_tree. You can use this function to print to the output an ASCII-based representation of any parse tree. This function is provided to help you debug your program.

The code for all these functions is in the file tree.c, but you do not have to look at the content of that file to do the homework assignment. Just use the functions provided in tree.h to build new parse tree nodes or to print parse trees, and you will be fine.

You do not have to look at or to understand the content of the header file varim.h, but it has to be included in your file parser.c. This header file simply contains some code to help with the simulation of function overloading in C.

The header file parser.h contains the descriptions of all the functions that you have to create in your file parser.c. So your file parser.c has to define the functions match_token, match_epsilon, match_B, match_L, match_D, match_S, match_E, match_Ep, match_T, match_Tp, and match_F. All these functions return a ptn, which means that all these functions must return a parse tree (which your parser has to build using the new_*_node function described in tree.h). By including the file parser.h in your file parser.c, the compiler will be able to check that your parser code correctly returns the right type of data structures, so this again will help you debug your code.

WARNING: you have to create a single file parser.c, which must include the four header files tokens.h, tree.h, varim.h, and parser.h. Do NOT modify any other file apart from parser.c.

To do the homework assignment, you need to do the following:

- download the hw03.zip file containing the files debug.h, lexer.l, main.c, parser.h, tokens.h, tree.c, tree.h, utils.c, utils.h, varif.h, and varim.h that you need to have to do the homework assignment.

- implement the file parser.c as described above.
  WARNING: do not use global variables in your code, use local variables

only, or you will lose points!

- compile the code:
    flex lexer.l
    gcc -o hw03 main.c lex.yy.c parser.c tree.c utils.c

(replace flex with lex if you only have lex on your computer).

You then should get an executable program called hw03. To test the program, run it and give it some sample input. The program should then print the corresponding parse tree. For example, for the input {}:

```
$ ./hw03
{}
    B
    |
+---+---+
|   |   |
|   |   |
{   L3  }
    |
    +
    |
    |
   eps
```

Note that, when you give input interactively to the program, the program will wait for more input until it sees a special 'End Of Input' token that marks the end of the input. When running the program interactively you can indicate the end of the input by typing a newline ("Enter") followed by Control-d (the Control and d keys pressed at the same time).

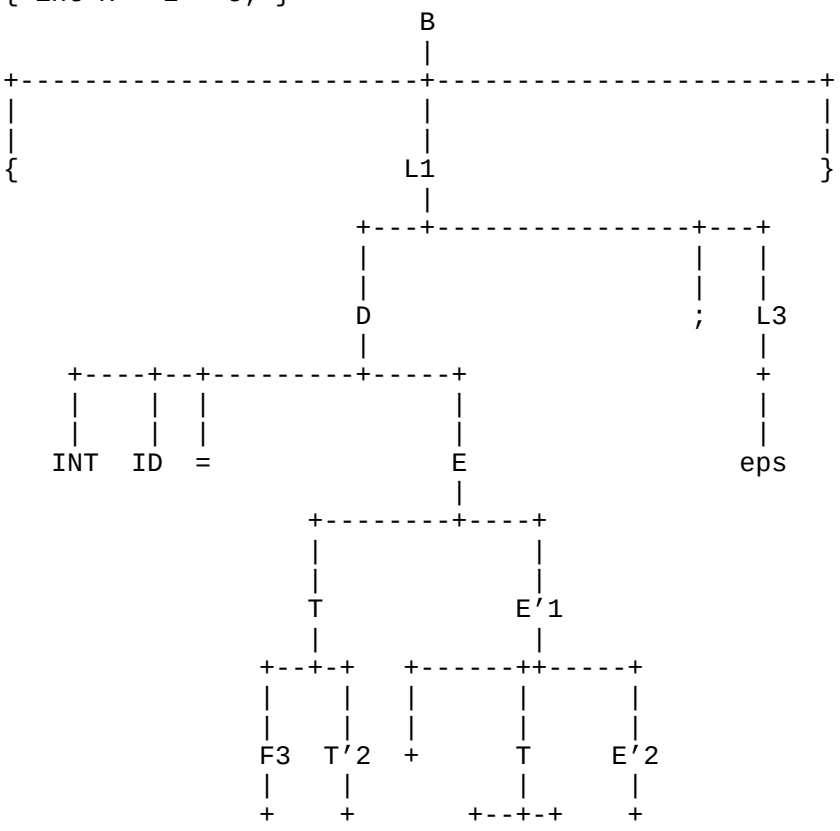
If you run the program and give it input from a file instead:

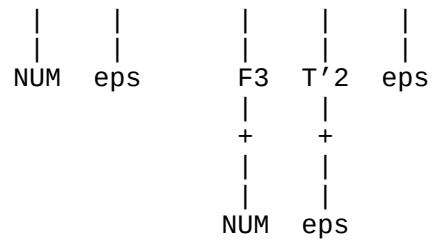
```
$ ./hw03 < input_file
```

then the program will automatically detect the end of the input when it reaches the end of the file.

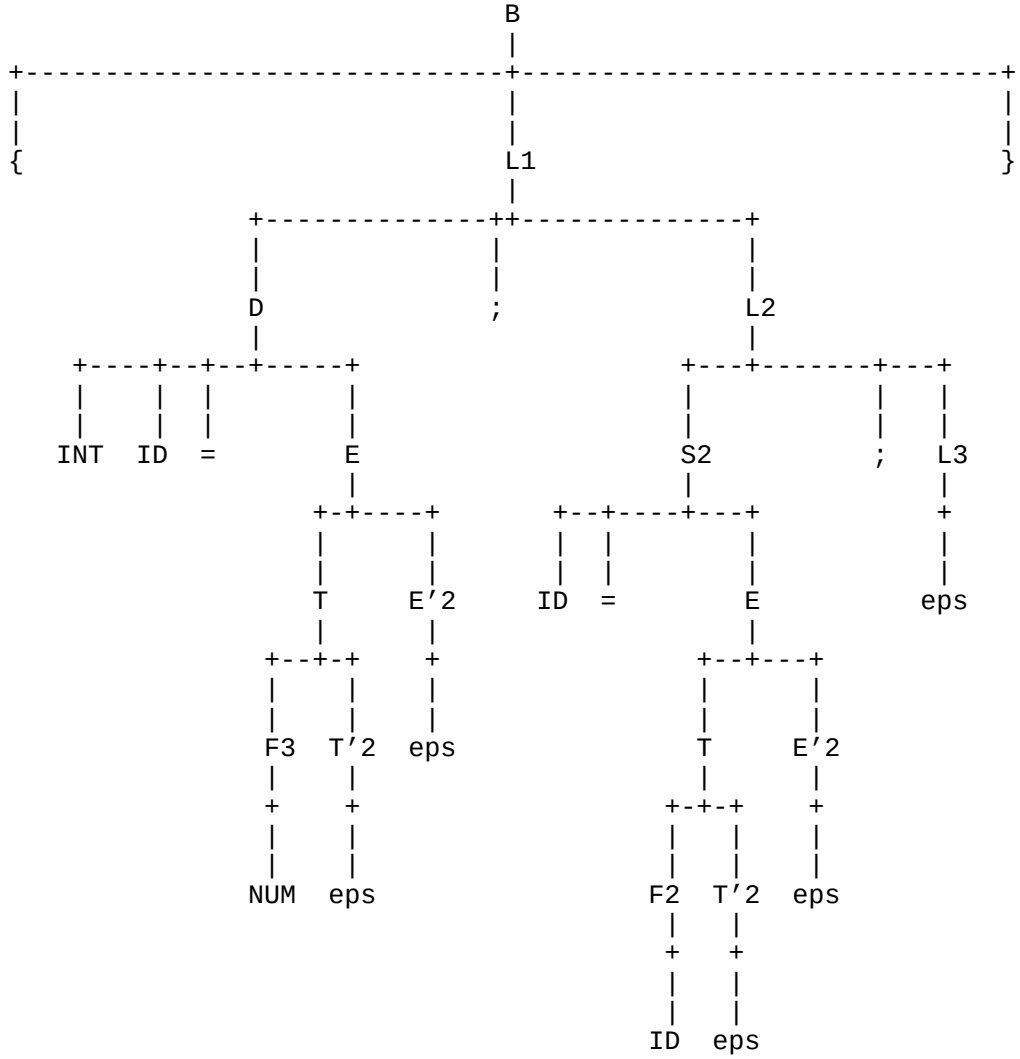
Here are two more examples:

```
$ ./hw03
{ int x = 2 + 5; }
```






```
$ ./hw03
{ int x = 2; y = x; }
```



Note that, when the program prints a parse tree, it does not print just the name of the nonterminals (like B, L, or F): if there is more than one grammar production for the same nonterminal then the program also prints the number of the grammar production that was used. For example, if the grammar production: F -> ID
is used to create a piece of parse tree, then, when printing the parse tree, the program will actually print "F2" for the root of that piece of parse tree, since the second grammar production for F was used. This is to help you debug your program.

Note also that the program has a special ’-d’ option that you can use when debugging your program, to get some basic information about which functions are called by your parser (where => represents a function call and <= represent a function return):

```
$ ./hw03 -d
{}
consuming input token: {
match_token => new_token_node at line 9 in parser.c
match_token <= new_token_node at line 9 in parser.c
peeking at lookahead token: }
match_epsilon => new_epsilon_node at line 16 in parser.c
match_epsilon <= new_epsilon_node at line 16 in parser.c
```

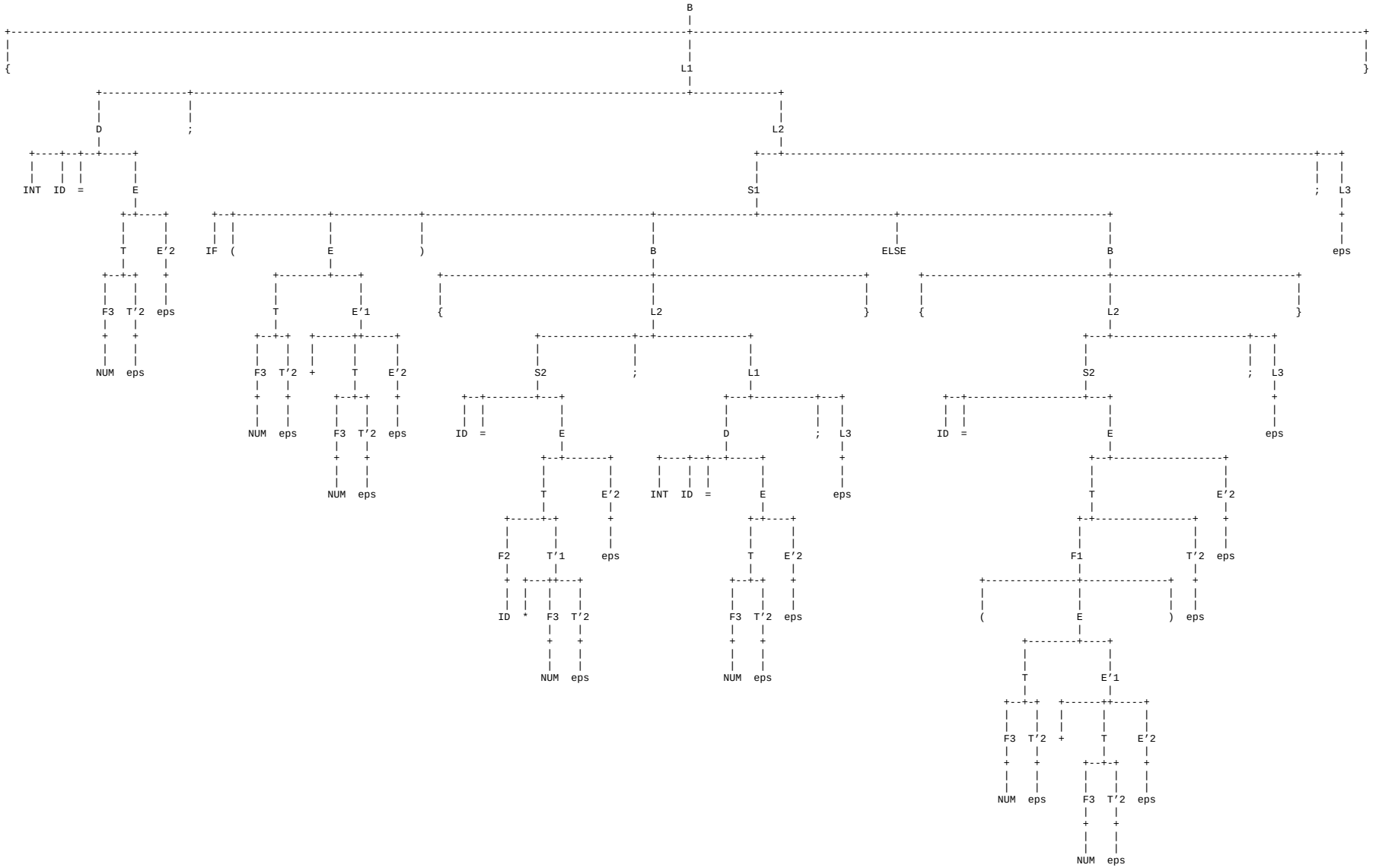
```
match_L => new_L_node at line 43 in parser.c
  new_L_node => new_L3_node at line 403 in tree.c
  new_L_node <= new_L3_node at line 403 in tree.c
match_L <= new_L_node at line 43 in parser.c
consuming input token: }
match_token => new_token_node at line 9 in parser.c
match_token <= new_token_node at line 9 in parser.c
match_B => new_B_node at line 24 in parser.c
match_B <= new_B_node at line 24 in parser.c
peeking at lookahead token: End Of Input
    B
    |
+---+---+
|   |   |
|   |   |
{   L3  }
    |
    +
    |
    |
   eps
```

Here is an example of using the program with syntactically incorrect input:

```
$ ./hw03
{ int x = ; }
syntax error, expected token ’NUM’ but found next input token ’;’ instead
```

Here is a bigger example to finish:

```
$ ./hw03
{ int x = 3; if (4 + 5) { y = y * 2; int z = 0; } else { a = (2 + 5); }; }
```


B
{
L1
}
D
;
L2
INT
ID
=
E
S1
;
L3
T
E’2
IF
(
)
B
ELSE
B
eps
F3
T’2
eps
NUM
eps
T
E’1
{
L2
}
{
L2
}
+
S2
;
L1
S2
;
L3
F3
T’2
NUM
eps
ID
=
E
D
;
L3
ID
=
E
eps
F3
T’2
eps
NUM
eps
T
E’2
INT
ID
=
E
eps
F2
T’1
eps
T
E’2
T
E’2
ID
*
F3
T’2
T
E’2
F1
T’2
eps
NUM
eps
NUM
eps
(
E
)
eps
T
E’1
F3
T’2
+
T
E’2
NUM
eps
F3
T’2
eps
NUM
eps