Language Processors
Final assignment - Complete
Translator of Expressions to a Language in Prefix Notation

Work to do:

1. Please read the entire statement before starting to work.
2. Rename your code file from previous sessions to **`trad4.y`** , and in subsequent sessions as **`trad5.y`** and so on. The final name should be **`trad.y`**
3. Develop the points indicated in the Specifications as far as you have time. In the next sessions we will continue with them.
4. Install the **`clisp`** interpreter in your *Unix* system using (for *Ubuntu*)

   ```
   sudo apt-get install clisp
   ```

   Test your program using:

   Option 1:

   ```
   ./trad <test.c | clisp
   ```

   Option 2:

   ```
   ./trad <test.c >test.l
   clisp test.l
   ```

   Option 3:

   ```
   ./trad <test.c >test.l ; clisp test.l
   ```

5. Continue with the work at the appropriate point, try to save a version of your code every time you solve a specification.
6. THE WEB INTERPRETER WILL NOT BE USED IN THE EVALUATION. Try to use it only for specific tests.

   `https://rextester.com/l/common_lisp_online_compiler`. For this you can:

   a. Edit a file (for example, **`test.txt`**) with a series of expressions
   b. Execute **`cat test.txt | ./trad`**
   c. Copy the output and paste it into the interpreter window
   d. Compile and run with F8.

Delivery:

Upload the file **`trad.y`** with the work done so far.

Include two initial comment lines in the file header. The first with your names and group number. And the second with the emails (separated with a space).

Also deliver a document called **`trad.pdf`** with a brief explanation of the work done.

Before and after making the delivery, check that it works correctly and that it complies with the indications.

Specifications

It is recommended to approach each of the steps sequentially.

1. The new **yylex** function skips lines starting with #. Change the print symbol **#** to the **$** symbol. Also include the parentheses to enclose the function **$(<exp>);** it should be translated to **(print <exp>)** [5] .

   Eliminate the production that generates **statement ::= <exp>**

2. Adapt the grammar to translate **print** with multiple parameters. The input **$(<exp1>, <exp2>, <exp3>);** must be translated into **(print <exp1>) (print <exp2>) (print <exp3>)**, in the same order.

3. Include the definition of global variables in the grammar. The definition in C will be **int <id>;** [6] which must be translated to **(setq <id> 0)** . It is necessary to include a second parameter in Lisp to initialize the variables. In the case of the simplest C initialization, this value will be 0 by default.

4. Extend the grammar to contemplate the definition of a variable with assignment included: **int <id>=<const.>;** which must be translated into **(setq <id> <const.>)**. We use **<const.>** here to represent a constant numeric value. We will not consider expressions in these statements for now.

   > This corresponds to the definition of global variables. Remember that in C it is not allowed to assign expressions (with variables and functions) to a global variable in the statement in which it is declared, since the process has to be done at compile time. Expression evaluation is done at runtime.

5. In C **main** is the main procedure/function, it must have a reserved word in the corresponding table and must be linked to the **main** *token*. Extend the grammar to recognize the main function. It must allow the inclusion of statements within its body. Remember that we may force the existence of the **main** function by a good design of the grammar.

C	Lisp

int a ;
main () {
  $ (a + 1) ;
}
	(setq a 0)
	(defun main ()
	  (print (+ a 1))
	)

	(main) ; To execute the program

[5] Later, we will substitute it with the statement corresponding to 'printf(..)'.

[6] From this moment we start translating from C to Lisp.

6. Consider that the structure of a C program should be:
<Decl Variables> <Def Functions>. In theory it should be possible to interleave both types of definitions, but this can lead to conflicts in the parser. Therefore, we will follow a fixed structure. The statements defined by the grammar should only appear within the body of a function. Check the structure of your grammar. It should be very carefully designed and structured, trying to be as hierarchical as possible.

> There are several issues here. 1) You have to design a grammar that is as hierarchical or structured as possible. Non-Terminal names must be chosen carefully and must be representative and meaningful. This will avoid the appearance of errors typical of "tangled" or excessively complicated designs. 3) It is suggested to use a program structure that begins with the declaration of variables and then with the definition of functions. 4) The recommendation is to define the functions in reverse order of hierarchy, starting with the simplest functions and ending with the **main**. This will allow the translator to always have a reference to the functions that are used later because they have already been defined previously. If parent functions are defined first and lower-rank functions second, a compiler will need to make inquiries about the type of called functions that have not yet been defined, or use two passes to perform partial and conditional translations. The C compiler requires in these cases a prior declaration (prototype) of the functions. In Lisp, functions must be defined before they can be used. To avoid complications with the prototypes we will use C programs with the functions in reverse hierarchical order.
>
> It is suggested not to allow the mixture of variable declarations and function definitions to simplify the grammar and avoid conflicts. This option is not prohibited. It is simply not recommended to resort to it at the beginning of the practice.

7. Variables can be declared inside the body of the functions, which in both C and Lisp will be local variables. The same translation of global variables is used, but in this case, they must be generated in the code belonging to the function.

C	Lisp

`int a ;`
`main () {`
`  int a = 4 ;`
`  $ (a + 1) ;`
`}`
	`(setq a 0)`
	`(defun main ()`
	`  (setq a 4)`
	`  (print (+ a 1))`
	`)`

	`(main) ; To execute the program`

Care must be taken when including the definition of local variables in the grammar, to avoid conflicts with the definition of global variables. This translation will be expanded in later points.

8. Extend the grammar to accept multiple definition of variables with optional assignments: **`int <id1> = 3, <id2>, ..., <idk> = 1;`** which must be translated to a sequence of individual definitions **`(setq <id1> 3) (setq <id2> 0) ... (setq <idk> 1)`**. Try to ensure that the order in which the variables are printed corresponds to the order of the declaration. Pay attention that constant (numeric) values are assigned, not evaluable expressions. This is because in C it is not possible to evaluate such expressions for global variables at compile time.

   Local variables can be declared in C by alternating with general statements. It is also possible to declare them with an initialization in which the value of an expression is assigned, unlike in the case of global variables. We propose to declare them only at the beginning of the code block and restrict the assignment in the declaration to numeric values. This is not mandatory, it is a recommendation to simplify the grammar and avoid possible conflicts.

9. Remove the production **`Statement → Expression ;`** Statements consisting on single expressions are allowed in C, but we will not use them right now.

10. To print literal strings we propose using **`puts("Hello world");`** which will be translated into **`(print "Hello world")`**. Note that **`yylex`** detects input sequences between double quotes and uses the **`String`** *token*.

11. Replace the symbol **`$`** used to print by the reserved word **`printf`**. The format of the printout will be completed to accommodate the required format
    `printf(<string>, <expr1>, … , <exprN>) ;` translating it to
    `(print <expr1>) (print <expr2>), …, (print <exprN>)`. Please bear in mind that the content of the formatting string is complex and requires very elaborate processing to interpret all formats. Therefore, it must be recognized by the grammar although it is not really translated. To print literal strings, we will continue using **`puts`**.

12. The arithmetic, logical, and comparison operators in C can be combined without the programmer being able to appreciate particular nuances. In reality, they are operators of a different nature. The first ones return numeric values, while the logical and comparison operators return boolean values. It would make sense to treat them separately in the grammar, but that would need to be done with a lot of caution, since some conflicts might arise. The logical and comparison operators in Lisp are related to the C equivalents in the following table:

C	&&	||	!=	==	<	<=	>	>=
Lisp	and	or	/=	=	<	<=	>	>=

    Pay attention to the associativity and precedence defined for each new operator you include.
    http://web.cse.ohio-state.edu/~babic.1/COperatorPrecedenceTable.pdf

13. The control structure **`while ( <expr> ) { <code> }`** will be translated with the structure **`(loop while <expr> do <code>)`** . Pay attention that this C control statement does not require a separator ; in the end.

14. Translate the control structure **if(<expr>){ <code> }** into **(if <expr> <code>).** In Lisp, the if structure without the **else** ends with the last parenthesis. The if control structure with the else block **if (<expr>) { <codigo1> } else { <codigo2> }** is translated into **(if <expr> <codigo1> <codigo2>)**. This extension may cause conflicts in bison that have been studied in the masterclass, so you just need to solve them with a proper design of the grammar. It is highly recommended to use { } to encapsulate blocks. Remember that the if control structure does not end in a semicolon when curly brackets are used. This is an example of a if-then-else structure

C	Lisp

?	(defun test-if (flag)
	  (if flag
	  123
	  456)
	)

	(print (test-if T)) → 123 (print (test-if NIL)) → 456

This shows the functional nature of Lisp, which has no correspondence in C. In Lisp, expressions return a value. In C such values must be assigned to a variable. You will not be able to directly use the Lisp functional notation when translating from C. Example of translation from C to Lisp:

C	Lisp

main () {
  int a = 1 ;
  int b ;
  if (a == 0) {
  b = 123 ;
  } else {
  b = 456 ;
  }
  printf (¨%d¨, b) ;
}
	(defun main ()
	  (setq a 1)
	  (setq b 0)
	  (if (= 0 a)
	  (setq b 123)
	  (setq b 456)
	  )
	  (print b)
	)

	(main)

To include several statements in the then or else branch of the if in Lisp, it is necessary to insert a function **progn** that receives these statements as parameters. This is illustrated in the function example below:

```
is_even (int v) {
  int ep ;
  printf ("%d", v) ;
  if (v % 2 == 0) {
      puts (" is even") ;
      ep = 1 ;
  } else {
     puts (" is odd") ;
     ep = 0 ;
  }
  return ep ;
}
```

```
(defun is_even (v)
   (setq ep 0)
   (print v)
   (if (= (mod v 2) 0)
     (progn (print "is even")
            (setq ep 1))
     (progn (print "is odd")
            (setq ep 0)) )
    ep
}
```

In C, conditionals are defined as generic expressions, without distinguishing between arithmetic, boolean and distinction between arithmetic, boolean and comparison expressions. One reason is that in C there is no boolean type, it is simulated by integer values. The boolean False value is simulated with an integer 0, and the True value with any other integer value. One problem is that conditionals of integer type are not supported in Lisp, they cause an error. The initial solution we are going to adopt initially is to assume that the test programs will use a canonical C style, without extravagances like **`while (10-a)`**. ...

Examples:

C	Lisp

`int a = 1 ;`
`main () {`
`  a = a && 0 ; // a = 0 ;`
`  while (10-a) { // 10-a != 0`
`  printf (¨%d¨, a) ;`
`  a = a + 1 ;`
`  }`
`}`
	`(setq a 1)`
	`(defun main ()`
	`  (setq a (and a 0))`
	`  (loop while (- 10 a) do`
	`  (print a)`
	`  (setq a (- a 1))`
	`))`

`int a = 1 ;`
`main () {`
`  a = 0 ;`
`  while (a != 10) {`
`  printf (¨%d¨, a) ;`
`  a = a + 1 ;`
`  }`
`}`
	`(setq a 1)`
	`(defun main ()`
	`  (setq a 0)`
	`  (loop while (/= a 10) do`
	`  (print a)`
	`  (setq a (+ a 1))`
	`  )`
	`)`

	`(main) ; To run the program`
`It will print the sequence:`	`0 1 2 3 4 5 6 7 8 9`

It is understood that a priori we will not use C programs that contain sentences such as those marked in yellow.they must be like those highlighted in cyan.

15. Implementation of the for control structure of C. We propose to restrict its use to its most canonical version. That is:
`for ( <initialization> ; <expr> ; <inc/dec> ) { <code> }` where **`<initialization>`** is a statement that assigns a value to the index variable, **`<expr>`** is a conditional expression whose (logical) value determines whether the loop continues or not, and **`<inc/dec>`** is a single statement to modify the value of the index variable. Use the structure **`(loop while <expr> do <code>)`** to translate it. Bear in mind that in the for control structure, the index is updated at the end of the code within its body. Of course it is possible to use this control structure in a not that strict way; for example, it would be possible to use one or more statements in **`<initialization>`** , more than one operation in **`<expr>`**, and even including the whole body within the third field. However, it is recommended to stick to the restricted version for this assignment. Do not implement operations of the type **`i++, ++i, i+=1;`** etc.

16. Array Implementation.

a. To declare a Array we can use a Lisp syntax extending the one we use for variables: if in C we define **`int myvector [32] ;`** we will translate it to: **`(setq myvector (make-array 32))`** creating an array named **`myvector`** with 32 elements. **We will not consider the option of initializing the elements.** In both C and Lisp the first element will be indexed with 0.

b. To access an element of an array and to operate with it within an expression we will use the function **`aref`**. For example, if we want to print the fifth element of the array **`myvector`** previously defined: **`printf ("%s", myvector[4]) ;`** should be translated as **`(print (aref mivector 4))`**. The function **`aref`** uses two parameters, the first being the vector itself, and the second one the index.

c. To modify an element of a vector: **`myvector [5] = 123 ;`** we must use the assignment function: **`(setf (aref myvector 5) 123)`**. In some Lisp variants you can use the function **`setq`**, but it is more common to use **`setf`**. This requires a differentiated treatment in the semantics for arrays.

Below you can find a sequence of Lisp instructions and the associated results once they are evaluated..

Lisp	Result
`(setq b (make-array 5))`	
`(print b)`	`#(nil nil nil nil nil)`

`(setq i 0)`
`(loop while (< i 5) do`
`(setf (aref b i) i)`
`(setq i (+ i 1))`
`)`	

`(print b)`	`#(0 1 2 3 4)`
`(setf (aref b 0) 123)`	
`(print (aref b 0))`	`123`
`(print b)`	`#(123 1 2 3 4)`
`(print (aref b 10))`	`Aref Index out of range`

17. Functions. Things to be taken into account:
    a. Some initial simplifications
        i. It is not necessary that functions return values (no **`return`** statement). In C we can write functions that do not return anything, even if they have a type defined (although we can consider it a malpractice). Even if they have a return, the function that calls them can ignore that value. In other words, we will be using functions as procedures.
        ii. We will be using explicit types for all functions. In C, functions that are defined as such are implicitly assigned type **`int`**. In our case, this option is useful, since otherwise the definition of functions and variables would have the same initial sequence of symbols: **`<type> <symbols>`**. This may cause conflicts, which could be solved with a proper rewriting of the grammar.
        iii. No parameters for now.
    b. Subsequently we may extend it so that function contains:
        i. A single parameter. It is understood that it should contain either zero or one parameter. This implies the grammar needs to be extended to define and call functions.
        ii. A list of arguments, which may include zero, one or more parameters. Special attention must be paid to ensure that the translated sequence of parameters follows the same order on both sides.
    c. Add the return statement:
        i. A well-structured function should only have the return statement as the last statement of the function; **`myfunction () {... return <expr> ;}`** should be translated to Lisp as **`(defun myfunction () ... <expr>)`**. It is worth remembering that **`return`** in C is followed by an expression that does not necessarily have parentheses. We do not consider it as a function.
        ii. However, in C, **`return`** statements can be placed anywhere within the function, although this is a bad practice in well-structured programming. For this, we will have to translate the statement **`return expression;`** as **`(return-from <function-name> <expression>)`**.
    d. Recall that in C, functions can be used either as procedures, ignoring the returned value, or as functions. The first case should be considered malpractice if the function returns some value. But we can also consider it as an additional section. That is, we can consider that functions, in addition to intervening in expressions, as a parameter to call another function, or in an assignment, may also be a statement in which only the function is called.

Examples of functions with one parameter and return:

```c
#include <stdio.h>

square (int v) {
    return (v*v) ;
}

factorial (int n) {
   int f ;
   if (n == 1) {
      f = 1 ;
   } else {
     f = n * factorial (n-1) ;
   }
   return f ;
}

is_even (int v) {
   int ep  ;
   printf ("%d", v) ;
   if (v % 2 == 0) {
      puts ("is even");
      ep = 1 ;
   } else {
      puts (" is odd") ;
      ep = 0 ;
   }
   return ep ;
}

main () {
   printf ("%d\n", square(7)) ;
   printf ("%d\n", factorial (7)) ;
   printf ("%d\n", is_even (7));
   is_even (8);
}
```

```lisp
(defun square (v)
    (* v v)
)

(defun factorial (n)
     (setq f 0)
     (if (= n 1)
        (setq f 1)
        (setq f (* n (factorial (- n 1))))
      )
      f
)


(defun is_even (v)
   (setq ep 0)
   (print v)
   (if (= (mod v 2) 0)
      (progn
        (print "is even")
        (setq ep 1))
      (progn
        (print "is odd")
        (setq ep 0)) )
   ep
)

(defun main ()
    (print (square 7))
    (print (factorial 7))
    (print (is_even 7))
    (is_even 8)
    )
```

```
(main)
→ 49
→ 5040
→ 7 is odd 0
→ 8 is even
```

Before addressing the next point, it is recommended that you carefully ensure that your practice is fully functional for the previous specifications. Save that version.

18. Promotion between arithmetic and boolean expressions: As already stated, in some dialects of Lisp it is imperative that conditional expressions provide a Boolean value. The problem is that C is very lax in this sense, it allows you to use expressions like **`if (a-1)`** ...which provides an integer value that is interpreted in C as **`0:False, other:True`**. In some Lisp interpreters this will cause an error or a wrong output since the expression in **`(if (- a 1)`** … provides an integer value when a boolean is expected. The reverse situation happens in C statements like **`a = a + (a < b) ;`** (although it has dubious interpretation, the intention seems to increment **`a`** only in case it is less than **`b`**). In Lisp it will give an error or a wrong output since it is not possible to add 1 to **`nil`** or **`T`**.
    It is requested to add type conversion between conditional/logical and arithmetic expressions. For conditionals you can insert the operation **`(/= 0 <expr>)`**. To adapt a conditional/logical expression to an arithmetic one it is recommended to insert a different function or expression in Lisp.

Example:

In C	**In Lisp direct trans.**	**In Lisp with promotion**

`test (int a, int b) {`
`  if (a-b) {`
`  puts ("Different") ;`
`  } else {`
`  puts ("Equal") ;`
`  }`
`}`
 	
`main () {`
`  test (1, 1) ;`
`  test (1, 2) ;`
`  test (2, 1) ;`
`}`
	`(defun test (a b)`
	`  (if (- a b)`
	`  (print "Different")`
	`  (print "Equal")`
	`  )`
	`)`
	
	`(defun main ()`
	`  (test 1 1)`
	`  (test 1 2)`
	`  (test 2 1)`
	`)`
	
	`(main)`
		`(defun test (a b)`
		`  (if (/= 0 (- a b))`
		`  (print "Different")`
		`  (print "Equal")`
		`  )`
		`)`
		
		`(defun main ()`
		`  (test 1 1)`
		`  (test 1 2)`
		`  (test 2 1)`
		`)`
		
		`(main)`

`Equal`
`Different`
`Different`
	`Different`
	`Different`
	`Different`
		`Equal`
		`Different`
		`Different`

Evaluate the possibility of using some additional attribute to detect the necessary conversions between types.

Before addressing the next point, it is recommended that you carefully ensure that your practice is fully functional for the previous specifications. Save that version.

19. Multiple assignments (we exclude arrays). In some languages it is possible to perform multiple assignments in a single statement. For example, in python variables can be assigned as follows: `a,b,c = x,y,z ;` . We propose to extend our C language including this option:

    a. The implementation in Lisp would consist of translating `a, b = b, a ;` into `(setf (values a b) (values b a))`. As a curiosity, in this case a swap is performed without requiring a temporary variable. Pay attention to the grammatical design. Expand to any number of variables.

    b. Another possibility is to define functions with multiple returns. For example:

Extended C	**Lisp**

`int swap (int a,int b) {`
`  return b, a ;`
`}`
	`(defun swap (a b)`
	`  (return-from swap (values b a))`
	`)`

`a, b = swap (a, b) ;`	`(setf (values a b) (swap a b))`

Tests	**Output**
`(setf a 1)  (setf b 2)` `(print a)  (print b)`	`1` `2`
`(setf (values a b) (swap a b))` `(print a)  (print b)`	`2` `1`
`(setf (values a b) (values b a))` `(print a)  (print b)`	`1` `2`

Before addressing the next point, it is recommended that you carefully ensure that your practice is fully functional for the previous specifications. Save that version.

20. Local scopes. The system we use to differentiate global from local variables does not work correctly, because **`setq`** and **`setf`** always define global variables, but in most of the tests provided you will not notice this. A scope system should ensure that local variables are only accessible within it, hiding global or higher-scoped ones with the same name. To implement these scopes, we will need to insert expressions with the **`let`** function.

    a. A single scope per function

C	Lisp

`main () {`
`  int a=1, b=2 ;`
 	
`  …`
 	
`}`
	`(defun main ()`
	`  (let ((a 1)`
	`  (b 2))`
	
	`  ...`
	`  ) ; closing the let`
	`)`

    b. Nested scopes. It is possible to make nested scopes, just like in C++.

C	Lisp

`main () {`
`  int a = 1 ;`
`  printf("%d\n", a) ;`
`  {`
`  int a = 2 ;`
`  printf("%d\n", a) ;`
`  }`
`  printf("%d\n", a) ;`
`}`
	`(defun main ()`
	`  (let ((a 1)) ; let #1`
	`  (print a)`
	`  (let ((a 2)) ; let #2`
	`  (print a)`
	`  ) ; closing let #2`
	`  (print a)`
	`  ) ; closing let #1`
	`)`

`(main)`	`1`
	`2`
	`1`

    c. Some of you like to program declaring local index variables in the **`for`** structure. The structure **`for (int i = 0 ; i < n ; i++ { int a ; … }`** must be translated by inserting at least one **`(let …)`** for the for code. We let the student decide the technical issues.

IMPORTANT ISSUES:

From this point onwards only some technical indications will be given for the specifications that are not clear.

Keep in mind that some new operators may appear in the tests that will have to be added to the grammar.

Contributions that have not been previously agreed with the teachers will not be considered.

In order to evaluate this practice, keep in mind that the difficulty will increase as the points raised progress. The first ones can be solved in class so they will also count less. In the last sessions your autonomy will be assessed when completing the proposed points.

It is not recommended in any case to leave the practice for the last minute. The practice sessions are intended to support errors that arise when using*bison*.

Delivery errors will influence the evaluation of the practice.

#1

The evaluation will use gnu Common Lisp in a Linux environment. We won't use the web interpreter.

There are other interpreters, but all of them can alter the execution conditions. Therefore, practices that only work in other interpreters cannot be accepted.

It should be available in guernika. But you can install it on your Linux system. On Ubuntu with the command:

```
sudo apt-get install clisp
```

The way to test would be with the following commands:

Option 1:

```
./trad <test.c | clisp
```

Option 2:

```
./trad <test.c >test.l
clisp test.l
```

Option 3:

```
./trad <test.c >test.l ; clisp test.l
```

Check that you have the appropriate PATH defined or use the path explicitly.

The output obtained with all options **should be the same** as you would get by compiling test.c with gcc and executing the result. The exception will be line breaks introduced by Lisp `print` function, and in some additional cases such as multiple values, which in standard C do not exist.

#2

Check in the **`yylex`** function the embed code directive **`//@`**. It will have a fundamental role in the evaluation tests.

To launch the Lisp code it is necessary to resort to an instruction //@ (main) which will be included at the end of the test program.

Example:

```
int a = 10 ;
int b = 23 ;

f1 (int a, int b) {
  return a+b ;
}

//@ (print (f1 2 3))      ; allows to test f1 from source code

main () {
  printf ("%d", f1 (a, b)) ;
}

//@ (main)          ; allows launching the execution of the program
```

It is forbidden to explicitly output a **`(main)`** in translation as this causes double executions and other problems. Also, do not include calls to **`exit()`** when you consider the program has finished.

#3

It is important to try NOT to print the Lisp translation in the axiom production. As explained in class, this presents several problems, including exponential consumption of dynamic memory.

You should try to print the translations, for example, every time you finish defining a function, etc.

Another undesired effect in case of printing at the axiom level will be that the embedded code directives will be transcribed before the translation is printed, which will cause errors.

#4

If you want to generate formatted Lisp output, you shouldn't try it from within the parser semantics, as that complicates and makes it difficult to understand. The solution consists of programming a specific backend function that receives the translated string to print. This backend function should traverse the Lisp expressions in the string and generate the desired format at the same time that it prints it.

#5

You must format the bison code so that it is readable. It is always a good practice, just like when we program. In AG you will find an example of good practices programming with ***bison***.

#6

Issues that will be considered in the evaluation:

- Errors in the initial and advanced tests (-½ per section of the statement specifications).
- Errors in the grammatical design that allow unexpected inputs to be processed, i.e. inputs that are not indicated in the specifications or that do not make sense.
- The use of left recursion except in those cases where it is necessary to use precedence and operator associativity with double recursion.
- Failures not detected by the tests provided by the student.
- Tests that crash the **`trad`** program or the Lisp interpreter.
- Shift-reduce or reduce-reduce conflicts.
- Code difficult to understand due to improper formatting or unsolicited additions (Lisp formatting of parentheses).
- Unauthorized modifications with provided code (**`yylex`**, etc.).