Language Processors
Final assignment - Fourth Part

Frontend Translator: Subset of Language C to LISP (intermediate code)

Backend Translator: LISP to Postfix Notation (final code)

In this fourth part the objectives are:

- to finish the frontend translator
- to start the backend part

Work to do (Frontend):

1. Please read the entire statement before starting to work.
2. Rename your code file from previous sessions to **`trad4.y`**.
3. Continue with the work at the appropriate point, try to save a version of your code every time you solve a specification.
4. Develop the points indicated in the Specifications as far as you have time.
5. Download the tests.zip archive that contains a series of C programs to test your translator.
6. Install the **`clisp`** interpreter in your laptop using
   ```
   sudo apt-get install clisp
   ```
   Test your program using:
   Option 1:
   ```
   ./trad4 <test.c | clisp
   ```
   Option 2:
   ```
   ./trad4 <test.c >test.l
   clisp test.l
   ```
   Option 3:
   ```
   ./trad3 <test.c >test.l ; clisp test.l
   ```
7. THE WEB INTERPRETER WILL NOT BE USED IN THE EVALUATION. Try to use it only for specific tests.

Delivery:

Upload the file **`trad4.y`** with the work done so far.
Include two initial comment lines in the file header. The first with your names and **team number**. And the second with the emails (separated with a space).
Also deliver a document called **`trad4.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.

ALSO CHECK PART II CONCERNING BACKEND (later in the document)

Additional Information:

#1

If you want to generate a formatted Lisp output (in what used to be known as pretty printing), you shouldn't try it from within the semantic actions of the parser, as that complicates the code and makes it difficult to understand.

A good solution is to program 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.

#2

Check the inline or embed code directive **`//@`** in the **`yylex`** function. It will play a key role in the evaluation tests. When **`yylex`** reads the **`//`** sequence it checks the next character. In case there is a **`@`** the lexer interprets this sequence as a special directive to transcribe the rest of the input to the output. Since the **`//@`** directive will be treated as a line comment in any C compiler we can use this functionality to include literal Lisp code inside our C programs.

A utility will be to launch the programs translated to Lisp. To do this we will use a **`//@ (main)`** instruction that will be included at the end of the test program.

Other utilities could be to test function calls or to print intermediate results.

Example:

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

f1 (int a, int b) {
//@ (print a)              ; allows to test values
  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
```

#3

Because of the above, you should avoid printing an explicit **`(main)`** when generating code as this will cause double executions and other problems.

Also, do not include calls to **`exit()`** when you consider the program has finished. This happens automatically when **`yylex`** reads an **`EOF`** and returns with **`return(0)`** to **`yyparse`**.

#4

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 when programming with ***bison***.

#5

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: (avoids extra output from the eval function)

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

Option 3: (avoids extra output from the eval function)

```
./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.

#6

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.

#7

Avoid printing messages or texts that are not part of the translation. If you want to print any error message, do it through the error channel stderr. But be aware that mixing it with the standard output (sdout) may interfere with corrections.

#8

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.).

Specifications for the FRONTEND: C subset to LISP

It is recommended to approach each of the steps sequentially.

The initial specifications for the provided trad1.y file are:

- There is a hierarchical structure starting with:
  - **Axiom**, takes care of recursion (**r_axiom**) and derives:
  - **Sentence**, which in turn handles assignments and printing expressions.
- The sentences containing an expression are eliminated. Although C allows sentences like **1+2;** we will not take advantage of them, and they can be a source of conflicts.
- You can add your solution for chained assignments if available.
- The solution uses deferred code. We will leave the possible use of AST for a later stage.
- Study the given solutions for:
  - **operand ::= (expression)**
  - **term ::= operand**
  - **term ::= + operand**
  - **term ::= - operand**

1. **Global Variables**: Include the definition of global variables in the grammar.

   a. The definition in C will be **int <id>;** [1] 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. There can be zero, one or more lines for variable declarations, each one starting with **int**.

   b. Extend the grammar to allow for the definition of a variable including initialisation: **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.

   c. Extend the grammar to accept multiple definition of variables with optional initializations: **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.

   > This corresponds to the definition of global variables. Pay attention that constant (numeric) values are assigned, not evaluable expressions. 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.
   >
   > Remember to differentiate variable declarations from code statements.

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

2. **Main procedure/function In C**: there should be a reserved word in the corresponding table 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`

   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.

   2) 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 to avoid 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.

3. **To print literal strings** we propose to use **`puts(<string>);`** which will be translated into **`(print <string>).`** For example **`puts("Hello world");`** is translated into **`(print "Hello world")`**. Note that **yylex** detects input sequences between double quotes and delivers it as a **`String`** *token*.

4. **Printing expressions and strings**: replace the symbol **@** used to print by the reserved word **printf**. The format of this function is
   `printf(<string>, <elem1>, <elem2>, ... , <elemN>);`
   a. Consider at first the simplified version with a single element:
      `printf(<string>, <elem1>);` translating it to **`(princ <elem1>)`**. Please bear in mind that the content of the first format **`<string>`** is complex and requires very elaborate processing to interpret all formats. Therefore, it must be recognized by the grammar although we won't translate it. The second parameter **`<elem1>`** can be either an **`<expression>`** or a **`<string>`** and is translated using **`(princ <elem1>)`**.
   b. Extend the grammar to allow for one or more arguments in the printf statement. **`printf(<string>, <elem1>, <elem2>, ... , <elemN>);`** should be translated to **`(princ <elem1>) (princ <elem2>), ..., (princ <elemN>)`**. All the parameters **`<elemX>`** can be either an **`<expression>`** or a **`<string>`** and are translated using **`(princ <elemX>)`** and preserving the original order. This is: **`(princ <elem1>)(princ <elem2>) . . . (princ <elemX>)`**

   > The **`print`** function outputs a newline before each output. The **`princ`** function omits this newline and the double quotes of strings.

5. **The arithmetic, logical, and comparison operators** in C can be combined without special considerations. 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:

   Pay attention to the associativity and precedence defined for each new operator you include.
   https://en.cppreference.com/w/c/language/operator_precedence

C	&&	||	!	!=	==	<	<=	>	>=	%	^
Lisp	and	or	not	/=	=	<	<=	>	>=	mod	expt

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

   And, what happens if the input contains something like **`while(10-a) { a=a+1 ; }`**? We will soon explain how to deal with these cases.

7. **IF control structure**: translate **`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>) { <code1> } else { <code2> }`** is translated into **`(if <expr> <code1> <code2>)`**. 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

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

To include more than one 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:

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

```
(setq ep 0)

(defun is_even (v)
   (princ 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 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 solution we are going to adopt is to assume **at first** that the test programs will use a canonical C style, without extravagances like **`while(10-a){...}`** .

Examples (we will use code like the highlighted in cyan rather than the yellow one).

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`
	`  (princ 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`
	`  (princ a)`
	`  (setq a (+ a 1))`
	`  )`
	`)`

`//@ (main)`	`(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.

8. **FOR control structure**. 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.
> Use increments in assignment of the type: **`i=i+1`** or **`a=a-2`**.

Example of a for loop in C in canonical style:

```
for (a = 0  ; a < n ; a = a + 2) {
    . . .
}
```

9. **Local Variables** can be declared inside the body of the functions, which in both C and Lisp will be local variables. A translation similar to that of global variables is used, but in this case they must be generated in the code belonging to the function. Care must be taken when including the definition of local variables in the grammar, to avoid conflicts with the definition of global variables. Carefully review the code in the example below to understand what is required in the following sections:

   a) Global variables will be defined as usual using **`setq`**

   b) Local variables will be defined using **`setq`** with a change, concatenating their name with the name of the function. This is due to the fact that variables defined or modified with **`setq`** are always global variables in Lisp, check the example below. Using the concatenation serves to avoid collisions between local and global variables.

Example in Lisp

`(setq a 1)`
`(defun mifun ()`
`  (setq a 2)`
`  (print a))`
 	
`(defun main ()`
`  (print a)`
`  (mifun)`
`  (print a))`

`(main)`
`1`
`2`
`2  ; lost initial value`

   c) Change the output code for assignment statements: use from now on **`setf`** rather than **`setq`**. Both functions **`setq`** and **`setf`** are very similar in Lisp, but we will use them in order to differentiate between variable declarations and assignment statements.

C	Lisp

`int a ;`
`main () {`
`  int a = 4 ;`
`  a = a + 1 ;`
`  printf ("%d", a+1) ;`
`}`
`//@ (main)`
	`(setq a 0)`
	`(defun main ()`
	`  (setq main_a 4)`
	`  (setf main_a (+ main_a 1))`
	`  (princ (+ main_a 1))`
	`)`
	`(main) ; To execute the program`

Use the underscore _ to concatenate names instead of the hyphen - . The hyphen is too ambiguous with respect to the unary sign or the subtraction operator.

To decide which variables used within a function should be concatenated with the function name, it is necessary to use a local table in which all locally declared variables are inserted. If a variable is not found in this table it can be deduced that it is a global variable and should not be concatenated. As a starting point, study the code of the **`keywords`** table and the **`search_keyword()`** function.

10. **Functions**. Things to be taken into account:
    a. Some initial simplifications
        i. **No return statement**. It is not necessary that functions return values . 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. **Implicit function types**: we will be using implicit types for all functions. In C, functions that are defined without a type specifier 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 tokens: **`<type> <identifier>`**. This may cause conflicts, which should be solved with a proper rewriting of the grammar.
        iii. **No parameter**s 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:

```
#include <stdio.h>

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

fact (int n) {
   int f ;
   if (n == 1) {
      f = 1 ;
   } else {
     f = n * fact (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)) ;
   puts (“ “) ;
   printf ("%d\n", fact (7)) ;
   puts (“ “) ;
   printf ("%d\n", is_even (7)) ;
   puts (“ “) ;
   printf ("%d\n", is_even (8)) ;
   puts (“ “) ;
   is_even (8) ;
}
//@ (main)
```

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

(defun fact (n)
     (setq fact_f 0)
     (if (= n 1)
        (setf fact_f 1)
        (setf fact_f (* n (fact (- n 1))))
      )
      fact_f
)


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

(defun main ()
    (princ (square 7))
    (print " ")
    (princ (fact 7))
    (print " ")
    (princ (is_even 7))
    (print " ")
    (princ (is_even 8))
    (print " ")
    (is_even 8)
)
(main)
```

```
⇒ 49
⇒ 5040
⇒ 7 is odd 0
⇒ 8 is even 1
⇒ 8 is even
```

11. **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. The size of the vector shall be an integer value.

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 **`(princ (aref mivector 4)).`** The function **`aref`** uses two parameters, the first being the vector itself, and the second one the index. The index may be an expression.

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`**. The index may be an expression.

d. They may be declared as global or local variables. We exclude the declaration of vectors as parameters of a function. That is, we do NOT contemplate a C declaration such as: **`int mifunc (int vect [7]) { ... }`**. When calling a function we can pass it an indexed element of a vector, it will behave as an individual variable: **`a = max (v[i], v[i+1])`**.

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

Lisp	Output
`(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)`
`(setf 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`

Backend Translator: LISP to Postfix Notation (final code)

We start with the second part of the translator that will address the design of a small backend to translate from Lisp to a low-level language for a stack machine (as an alternative to the more conventional cpus programmed with Risc V or x86 assembly language).

Work to do (Backend):

1. Please read the entire statement before starting to work.
2. Rename your code file from previous sessions to **`back4.y`**.
3. Continue with the work at the appropriate point, try to save a version of your code every time you solve a specification.
4. Develop the points indicated in the Specifications as far as you have time.
5. Download the tests.zip archive that contains a series of C programs to test your translators (frontend and backend).
6. Install the **`gforth`** interpreter in your laptop following the instructions in the section *Resources for carrying out the Assignment*.
7. Your translators will be evaluated using the following work pipeline
   ```
   ./trad4 <test.c | ./back4 | gforth
   ```
   THE WEB INTERPRETER WILL NOT BE USED IN THE EVALUATION. Try to use it only for specific tests.
8. Ensure that gforth delivers the same results as using
   ```
   ./trad4 <test.c >test.l ; clisp test.l
   ```
   Or **`./test`** after compiling **`test.c`** with **`gcc`**.

Delivery:

Upload the file **`back4.y`** with the work done so far along with the frontend file.
Include two initial comment lines in the file header. The first with your names and **team number**. And the second with the emails (separated with a space).
Include in the report **`trad4.pdf`** 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.

Brief Introduction to Forth

Although the goal of the final lab is to create a translator from C to intermediate code, we are going to start with a more elementary phase: translating the expressions that we evaluate with our calculator into code that can be interpreted in Forth.

That is, it is necessary to translate expressions like

2*3<intro>
A=2*3<intro>
B=A*10+23/10<intro> etc.

in code that is evaluable by a Forth interpreter.

It is recommended to use a Forth interpreter to test the examples that are presented here. Check the last page to install **gforth**, or alternatively use the online interpreter: http://nhiro.org/learn_language/FORTH-on-browser.html

The main peculiarity of Forth is that it uses a stack of parameters on which all functions operate. Thus, the multiplication operation * will take the top two values in the stack, and replace them with their product.

```
| 3 |            |  |
| 2 |  →         | 6 |
 ---              ---
```

In notation of the so-called *stack comments*, for the multiplication we would have:

(a b – a*b)

To evaluate the expression**2*3<intro>** in Forth we will use the following sequence:

2 3 * . <intro>

The numbers 2 and 3 are recognized as such (*literals*) and are placed by the shell on the parameter stack. The operator (*word*) * extracts them and replaces them with the product of both. The operator . is a *word* which prints the value from the top of the stack (removing it). It is easy to recognize that the infix notation we are used to (binary operator between operands) has been transformed into postfix notation (Reverse Polish Notation: first the operands, then the operator). Anyone who has handled old HP calculators will be used to this notation. Note that between each of the input symbols there must be at least one blank space. The line is evaluated by pressing ***<intro>***, and the interpreter will respond with:

6 ok

Other arithmetic operators are: +, - and / . They all operate in the same way as the multiplication. It is important to specify that the numbers in the stack are always integers, generally with the range of values -32768...+32767. The division will therefore be integer and has its complement in the operator **mod** which calculates the remainder of a division. To negate a number there is the word **negate (a -- -a)**.

Some common stack words are (among others):

- **dup (a -- a a)** duplicates the value at the top of the stack
- **swap (a b -- b a)** swap the top two values
- **drop (a b -- a)** remove the value from the top

Any of the operations performed on a stack that has fewer operands than necessary will cause an error and, in some interpreters, the stack will be flushed.

To define a variable we will use the *word* **variable**, classified in Forth as a *constructor word*, because it adds a definition to the internal *dictionary*. To create variables A and B:

```
variable A variable B
```

To perform the operation **A=6**, we will use the operator **!** .

```
6 A !
```

The interpreter identifies the letter A as a *word* defined in the *dictionary* as variable. So it leaves the memory address assigned to the variable on the stack. The operator ! takes an address and a value from the stack, and stores that value in the address. In a similar way we can solve **A=3*2** :

```
3 2 * A !
```

To retrieve the value of a variable, we will use the operator **@** :

```
A @ .                    will print:
6 ok
```

The expression **B=A*10+(23/10)** should be translated to Forth as:

```
23 10 / A @ 10 * + B !
B @ .                          will print:
62 ok
```

Although it is not necessary right now, we can give a little insight into the philosophy of this language. To build new *words* of type function, we will use another *constructor*, the *word* : (colon).

```
: dup square * ;
```

Here we define the word square containing the sequence dup * and the definition terminator ; (semicolon) . We have an application of this defined word in the example:

```
5 square .          which is evaluated with
25 ok
```

In C we would have the equivalent definition:

```
int function square(int value)
{
      return (value*value) ;
}
```

Forth allows the use of both the **If-Then-Else** and **While** structures. **Their use is only allowed inside the definition of a word (function).**

The **If-Then-Else** version with the **Else** branch is:

```
[condition] IF
    [code to execute if condition is true]
ELSE
    [optional: code to execute if condition is false]
THEN
```

- **[condition]** is a boolean expression that should leave either a true (non-zero) or false (zero) value on the stack.
- **IF** checks the top of the stack. If it's true, the code between **IF** and **ELSE** (if present) is executed. If false, execution skips to **ELSE** (if present) or **THEN** .
- **ELSE** is optional and provides an alternative branch if the condition is false.
- **THEN** marks the end of the **IF** block.

Example:

```
: test-condition ( n -- )
    dup 0 >= IF
        ." Positive number" cr
    ELSE
        ." Non-positive number" cr
    THEN ;
```

The version without the **ELSE** branch is:

```
[condition] IF  [code to execute if condition is true]  THEN
```

The **WHILE** structure can be implemented with the loop **BEGIN ... WHILE ... REPEAT**

```
BEGIN
    [condition]
WHILE
    [code to repeat]
REPEAT
```

- **BEGIN** marks the beginning of the loop.
- **[condition]** is a boolean expression that should leave either a true (non-zero) or false (zero) value on the stack.
- **WHILE** checks the top of the stack. If it's true, the code between **WHILE** and **REPEAT** is executed. If false, execution leaves after the **REPEAT** label.
- **REPEAT** transfers execution to the **BEGIN** label.

Example:

```
: countdown ( n -- )     \ Expects a number in the stack
    BEGIN
        dup 0 >          \ Check if number is positive
    WHILE
        dup . cr         \ Print the number
        1 -              \ Decrement the number
    REPEAT
    drop ;               \ Drop the final zero from the stack
```

Additional Information:

#9

We will use FORTH as a CPU / basic stack machine alternative with a limited instruction set. This is why only specific FORTH instructions will be allowed. Specifically, all the previously mentioned:

- Arithmetic, Logic, and Comparison instructions
- `@` and `!` to handle variables.
- Stack operators `(drop, dup, swap, over, rot)`.
- `:` and `;` for function definition
- `IF, THEN, ELSE, DO, WHILE, REPEAT` for control structures.
- `. ." " cr` for printing and terminal output.
- `VARIABLE` for defining variables.
- `CELLS, ALLOT` for arrays.

Other words are not allowed, nor are Forth constructors.

#10

The evaluation of the assignment will depend on whether the frontend-backend concatenation generates the same results as expected from the original compiled C program or from the output obtained in Clisp from the frontend translation.

```
./trad <test.c | ./back | gforth
./trad <test.c >test.l ; clisp test.l
./test
```

after compiling `test.c` with `gcc`.

Resources to carry out the assignment:

Forth

The level of knowledge of Forth necessary for the practice is elementary, but we leave here the following links to serve as a reference:

- http://en.wikipedia.org/wiki/Forth_%28programming_language%29 introduction (in English)
- http://www.disc.ua.es/~gil/forth.pdf Introduction in Spanish that, beyond giving a vision of the basic operation, extends into the principles of programming.
- http://www.complang.tuwien.ac.at/forth/gforth/ gforth the GNU.
- http://www.complang.tuwien.ac.at/forth/gforth/Docs-html/ its documentation (in english and html)
- http://www.complang.tuwien.ac.at/forth/faq/faq-general.html set of FAQs about the language itself
- Online gforth interpreter: http://nhiro.org/learn_language/FORTH-on-browser.html

Instructions to install and get started with gforth:

In http://www.complang.tuwien.ac.at/forth/gforth/ The sources and binaries needed to install a Forth interpreter are available. Installation on Linux should be easy by following these steps:

- Download the file **`gforth-0.7.3.tar.gz`** (older versions should work too, but can give problems in some Linux distros).
- ~~Unzip~~ Untar with
  - ~~`gzip -d gforth-0.7.3.tar.gz`~~
  - ~~`tar xvf gforth-0.7.3.tar`~~
  - `tar xvf gforth-0.7.3.tar.gz`
- In the folder **`gforth-0.7.3/`** execute the configuration with **`./configure`**
- Compile the project with **`make`** and several interpreters/compilers are created.
- **`gforth`** can be used with the command **`./gforth`**. To exit the interpreter, you must use the *word* **`bye`**.
- You can move the **`gforth`** interpreter into another folder, do not forget to move also the image **`gforth.fi`**.

On Ubuntu you can use **sudo apt-get install gforth**
In case of error, try to run **sudo apt-get update** first.

If the installation gives you problems, use guernika for testing and discuss the problem with your instructors.

Specifications for the BACKEND: Translation from LISP to FORTH

Use bison for this part.

It is recommended to approach each of the steps sequentially.

1. Design the grammar for a basic LISP kernel. Add this design to the bison template (maybe **trad1.y**). This file already contains several functions (yylex, search_keyword, etc.) that can be useful. Try to design a direct translation parser without using deferred code.

2. Global variables are declared with **variable <var>**. You can assign an initial value using **<value> <var> !**
   Do not try to create local variables or arrays for now.

3. Generic functions and the main procedure are defined using **: <fname> <code> ;** or **: main <code> ;**
   Always be very careful printing spaces before and after each Forth operator.

4. For printing strings Forth uses the **."** word with another **"** to end the string. Example:
   ." Hello World"

   The space after the **."** is important, but the ending **"** does not need a previous space.

5. Printing an integer value is done with the **.** word.
   Examples:
   ```
   1 .
   A @ .
   ```

6. Assigning an integer value to a variable requires the use of the **!** (store) operator. Examples:
   ```
   1 A !          ( A = 1 ; )
   A @  B !       ( B = A ; )
   ```

7. The while structure is as follows:
   begin [expr] while [code] repeat
   which is equivalent to **while (<expr>) { <code> }**, where **[expr]** and **[code]** are the postfix translations for **<expr>** and **<code>**

8. The if structure is:
   [expr] if [code1] else [code2] then
   which is equivalent to **if <expr> then <code1> else <code2> endif**
   Note the postfix syntax (**if-else-then**). In Forth, the word **then** marks the end of the **if** block.
   For the case of an **if** without an **else** branch: **[expr] if [code1] then**

9. The correspondence between arithmetic, logical and comparison Operators is the following one:

C	&&	||	!	!=	==	<	<=	>	>=	%
Lisp	and	or	not	/=	=	<	<=	>	>=	mod
Forth	and	or	0=	= 0=	=	<	<=	>	>=	mod

In C the values of False and True are usually represented by 0 and 1. In Forth by 0 and