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

This second part of the practice continues extending the specifications to allow the translation of a small subset of C code. This will include conditional structures (if, while, for), functions and arrays. The output will still be in prefix notation, which corresponds to a variant of the Lisp language, which we will use as an intermediate language.

Future parts will also 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:

1. Please read the entire statement before starting to work.
2. Rename your code file from previous sessions to **`trad2.y`**.
3. Develop the points indicated in the Specifications as far as you have time. In future sessions we will continue with them.
4. Check your local installation of CLisp.
5. You can evaluate the translation results using the online interpreter `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 | ./trad2`**
   c. Copy the output and paste it into the interpreter window
   d. Compile and run with F8.

Delivery:

Upload the file **`trad2.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 **`trad2.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.

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`**.

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. 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. In C **main** is the main procedure/function, 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. 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.

   > 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

6. The control structure **while (<expr>) { <code> }** will be translated with the 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.