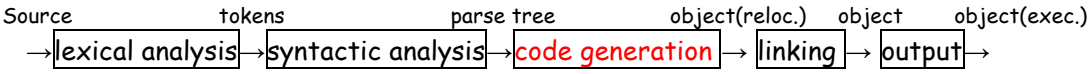
Práctica 3

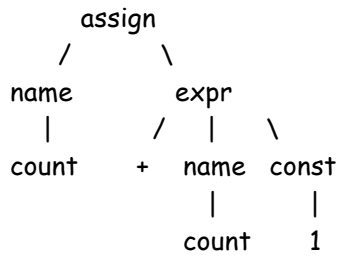
A simple compiler written in Prolog



Usually compilation is performed in five stages (see figure above). The 1st stage, lexical analysis is relatively uninteresting so we don't discuss it further. Práctica 1 and 2 (parsing) was related to the 2nd stage, syntax analysis. Essentially the effect of the analysis is to recognize the abstract program structure encoded in the characters of the source text and give this structure a name. For example, the name of the statement COUNT:=COUNT+1 may be

```
assign( name(count),expr(+,name(count),const(1)) )
```

which can be seen as the tree:



The third stage (the topic of this Práctica), code generation, produces the basic structure of the object program, but machine addresses are left in a symbolic form. These addresses are computed and filled by the fourth stage, assembly.

Thus, the assignment of this part (and next práctica) is to write a code generator for a small imperative language SMALL. The output of the code generator should be machine language of a one-accumulator computer. SMALL has **assignment**, **if**, **while**, **read** and **write** statements plus a selection of arithmetic (+, -,* and /) and comparison (=, <, and >) operators restricted to positive integers. The target language instructions are given bellow:

Arithmetic literal: ADDC, SUBC, MULC, DIVC, LOADC
Arithmetic memory: ADD, SUB, MUL, DIV, LOAD, STORE
Control transfer: JUMPEQ, JUMPNE, JUMPLT, JUMPGT, JUMPGE, JUMP
Input output: READ, WRITE, HALT.

As an example of the compiler function here is a simple program (to compute factorials):

```
READ VALUE;
COUNT:=1;
RESULT:=1;
WHILE COUNT < VALUE DO
      (COUNT:=COUNT+1;
      RESULT:=RESULT*COUNT);
WRITE RESULT
```

A straightforward translation into machine language, which the compiler will produce is:

Symbol	*Address*	*Instruction*	*Operand*
	1	READ	21
	2	LOADC	1
	3	STORE	19
	4	LOADC	1
	5	STORE	20
LABEL1	6	LOAD	19
	7	SUB	21
	8	JUMPGE	16
	9	LOAD	19
	10	ADDC	1
	11	STORE	19
	12	LOAD	20
	13	MUL	19
	14	STORE	20
	15	JUMP	6
LABEL2	16	LOAD	20
	17	WRITE	0
	18	HALT	0
COUNT	19	BLOCK	3
RESULT	20		
VALUE	21		

and for example, the name for LOAD 19, ADDC 1, STORE 19 may be :

```
(instr(load,19) ; instr(addc,1) ; instr(store,19))
```

where the ' ;' functor is only used to indicate sequencing.

Part 1: Compiling the Assignment Statement

Consider the assignment statement **Name** := **Expression**. The code for this will have the form:

```
Expr_code
STORE  Address
```

Where Expr_code is the code to evaluate the arithmetic expression **Expression** yielding a result in the accumulator. The STORE instruction stores this result at Address, the address of the location named **Name**. To make this precise we translate into a Prolog clause. The Prolog term which names the source form is:

assign(name(X),Expr)

where X and Expr are Prolog variables which correspond to **Name** and **Expression** above. Similarly, a Prolog term naming the target form is:

(E_code; instr(store,Addr))

where E_code and Addr are Prolog variables which correspond to Expr_code and Address. Suppose the source language names are to be mapped into machine addresses in accordance with a dictionary D. Then one necessary condition is expressed by the Prolog goal

lookup(X,D,Addr).

As you defined it in Práctica 2. The condition relating Expr and E_code may be expressed by the goal

encodeexpr(Expr,D,E_code)

with meaning: E_code is the code for the expression Expr conforming to dictionary D. If encodestatement(X,Y,Z) is a similar predicate meaning Z is the code for the statement X conforming to dictionary Y, write the definition for

encodestatement(assign(name(X),Expr),D,(E_code ; instr(store,Addr))) :- ...

Part 2: Compiling Arithmetic Expressions

The task here is to define the predicate:

encodeexpr(Expr,D,E_code).

The clauses for encodeexpr must translate the different types of arithmetic expressions:

Expression	*Instruction*	*Comments*
Const	LOADC Const	
Name	LOAD Addr	where Addr is the address of Name
Expr1 Op Expr2	Expr1code Instr	where Expr1code is the transaltion of Expr1 and Instr which applies Op

Here we will assume that Expr2 is either a constant or a variable (not a composite expression). The translations for the different operators is (the first one is applied to a constant and the second applied to a name):

Operator	*Instruction*
+	ADDC, ADD
-	SUBC, SUB
*	MULC,MUL
/	DIVC, DIV

As an example: Expr + 7 translates to Exprcode ADDC 7.
Sample outputs of the program so far may look like this:

```
?- encodestatement(assign(name(x),const(a)),D,X).
D = dic(x, _G521, _G525, _G526)
X = instr(loadc, a);instr(store, _G521)

?- encodestatement(assign(name(x),name(y)),D,X).
D = dic(x, _G515, _G519, dic(y, _G523, _G527, _G528))
X = instr(load, _G523);instr(store, _G515)

?- encodestatement(assign(name(x),expr(+,name(x),const(a))),D,X).
D = dic(x, _G632, _G636, _G637)
X = (instr(load, _G632);instr(addc, a));instr(store, _G632)
```

Submitting your answer

The práctica can be solved in teams of two people (1 submission for team). Submission is by email (to rafael.ramirez@upf.edu), the subject of the email must be *'P3- nombre1 apellido1- nombre2 apellido2*. and the message body should contain the programs for 1) and 2) and some tests to your programs. Deadline is 1 week later after the date of práctica 3. Late submissions will have a penalty.