2024

Caso de estudio
5. Utilizar el lenguaje Prolog para construir un compilador de un subconjunto de un lenguaje imperativo.

MANUEL CANO GARCÍA && ALEJANDRO CAÑAS BORREGUERO

Tabla de contenido

1.- Resumen de conceptos

Durante toda la implementación del caso de estudio elegido, hemos empleado conceptos adquiridos en la asignatura de "programación declarativa" para poder desarrollar el programa *prolog*.

Dentro de estos conceptos necesarios, hemos necesitado el **uso de listas**, para entre otras cosas, guardar los tokens del programa y poder ejecutar el programa a través de la maquina abstracta implementada. Una lista es una estructura de datos que se corresponde con la forma de **[X|Xs]** donde X es la cabeza de la lista, es decir el primer elemento de la lista y Xs es la cola de la lista, que correspondería con el resto de la lista siendo Xs una lista también.

Para leer el programa que querremos compilar, es crucial el uso de **elementos de Entrada/Salida** para archivos de texto, con la ayuda de predicados como **open** para abrir un fichero y establecer un flujo donde leeremos los caracteres gracias a predicados de la familia de **get** para posteriormente utilizarlos de la forma que nos convenga y el predicado **close** para cerrar el fichero correctamente.

Un concepto clave aplicado al crear los tokens de nuestro programa lo encontramos en el **uso de acumuladores**; pueden del tipo que se quiera y la principal característica de los acumuladores es que primero hay que inicializarlos (a un valor que no influya nada). Su forma de utilizarlo es bastante simple, que consiste en el caso recursivo en ir actualizando el valor de la variable acumulador y al llegar al caso base del predicado, "asignarle" el valor de la variable acumulador a la variable que queramos que guarde el resultado final.

El uso de la **operación corte** ha sido necesario para eliminar todas las posibles ramas de fallo del programa que no aportaban nada, para así aumentar la eficiencia del programa debido a que, al eliminar esas ramas del árbol de búsqueda, ahorramos a nuestro programa pasos de inferencias que no van a llegar a ninguna situación de éxito, a estos tipos de cortes se les llama **cortes verdes**, porque al usar el corte no estamos ignorando/perdiendo ninguna otra solución. Ligado a este concepto, también el uso del operador **->** para la implementación del **if-then-else** en *prolog.* La estructura que sigue el operador -> para su uso es:

(C -> A1; A2)

Sea C una condición en la que comprobaremos su veracidad o falsedad, si se cumple entonces ejecutamos A1, en caso contario ejecutaríamos A2.

También un concepto importante que influye en todo el programa es cómo *prolog* infiere dadas una serie de hecho y objetivos para llegar a un resultado, esto lo hace a través de **resoluciones SLD**, que, dado un objetivo principal, vamos construyendo el **árbol de búsqueda** dando pasos de resolución SLD, siguiendo una estrategia de profundidad con vuelta atrás hasta terminar con la ejecución del programa.

En el árbol de búsqueda, podemos identificar las ramas del árbol como **derivaciones SLD** al dar **pasos de resolución SLD**, donde una rama puede terminar en éxito **(Rama de éxito**) o en fallo (**Rama de fallo)**. Al dar los pasos SLD buscamos si se puede unificar el objetivo con un hecho, para producir la inferencia hasta llegar a la cláusula vacía (Rama de éxito) o no dar más pasos SLD porque el objetivo puede no unificar con ningún hecho (Rama de fallo).

2.- Documentación

El proceso de compilación de un programa de un lenguaje imperativo presenta varias fases diferenciadas, donde ejecutándose estas fases secuencialmente obtendremos la compilación de un programa junto con la ejecución de este. El primer paso para un compilador sería el proceso de **tokenizar** los elementos del programa, dichos tokens corresponderán a elementos claves del programa que deben ser reconocidos por el compilador a la hora de poder ejecutar el programa.

Una vez obtenidos los tokens del programa, el siguiente paso sería realizar el **árbol del programa** a través de gramáticas de cláusulas definidas, esto será la estructura utilizada por el programa *prolog* para poder ejecutar instrucción a instrucción del programa utilizando una **maquina abstracta** que entenderá el árbol.

2.1.- Tokeniser

El proceso de tokenizar el programa se lo encargamos al predicado que hemos definido como **`tokenizar(Tokens,Fichero)`** siendo Tokens la lista de tokens resultante tras la lectura del fichero y la variable Fichero el nombre del programa.

```
tokenizar(Tokens,Fichero) :- obtener_listado_car(Fichero,Lista_char),
                             procesar_tokens(Lista_char,Tokens).
```

Tenemos 2 predicados auxiliares que nos ayudan en la labor de tokenizar el programa:

- **`obtener_listado_car(Fichero,Lista_char)`**: Su función es básicamente convertir el fichero en una lista de caracteres, para luego gestionar carácter a carácter más tarde en el proceso de tokenización.

```
% Predicado para leer un archivo caracter a caracter
leer_caracteres(Stream, Lista) :-
    at_end_of_stream(Stream), !, % Verificar si hemos llegado al final del archivo
    Lista = []. % Si hemos llegado al final, la lista está vacía

leer_caracteres(Stream, [Caracter|Cola]) :-
    get_char(Stream, Caracter), % Leer el siguiente carácter como un atomo
    leer_caracteres(Stream, Cola). % Leer el resto de los caracteres


obtener_listado_car(Fichero,Lista_char) :- open(Fichero, read, Stream),
   leer_caracteres(Stream,Lista_char),
   close(Stream).
```

Lo primero abrimos el fichero con el predicado predefinido en el intérprete de *prolog* como **`open/3`** **`open(+SrcDest, +Mode, --Stream).`** Donde SrcDest indica el fichero que querremos abrir, Mode el modo, en nuestro caso como lo querremos para lectura optamos por la opción read y como resultado en el tercer argumento obtenemos un flujo de fichero que lo necesitaremos para ir leyendo el fichero de texto en el predicado definido como **`leer_caracteres/2.`**

Dicho predicado se encarga de ir leyendo carácter a carácter gracias al predicado **`get_char/2`** y lo iremos almacenando en una lista de caracteres, así hasta llegar al fin del archivo con el predicado **`at_end_of_stream/1`**.

- **`procesar_tokens(Caracteres,Tokens)`**: En este predicado, se efectuará todo el proceso de creación de los tokens respectivos gracias al predicado auxiliar **`concatenar_caracteres_al_reves/3.`** Su función reside en leer cada elemento de la lista de caracteres y dependiendo del carácter leído, realizar una serie de operaciones u otras.

```
concatenar_caracteres_al_reves([],[],[]) :- !.

%Caso base: ultimo token
concatenar_caracteres_al_reves([],Acumulador,[Token|Tokens]) :-
    reverse(Acumulador, Reversa), atomic_list_concat(Reversa, Token), concatenar_caracteres_al_reves([],[],Tokens).
```

Para empezar, tenemos el caso base, que correspondería con el último token a generar, esto se ve reflejado en el primer argumento con una lista vacía, ya que hemos terminado de leer todos los caracteres de nuestro programa. El caso de parada del programa se puede ver que todos sus parámetros resultan ser listas vacías ([]) junto con la operación de corte logramos construir correctamente la lista de tokens y cortar la generación de más ramas del árbol de búsqueda.

```
%Caso de ser un caracter especial (; , ( ,),...) monocaracter
concatenar_caracteres_al_reves([Caracter|Caracteres], Acumulador, Tokens) :-
    member(Caracter, [';',':','=','(',')','+','-','*','/',',','<','>','.']), !,
    reverse(Acumulador, Reversa),
    ( Reversa \= [] ->
        atomic_list_concat(Reversa, Atomo),
        ( atom_number(Atomo, Numero) ->
            Tokens = [Numero,Caracter|RestoTokens]
            ;
            Tokens = [Atomo,Caracter|RestoTokens]
        )
        ;
        Tokens = [Caracter|RestoTokens]
    ),
    concatenar_caracteres_al_reves(Caracteres, [], RestoTokens).
```

Si el carácter leído es especial, significa que estos caracteres por ellos mismos son tokens del programa, entonces damos la vuelta al acumulador para tener la secuencia de caracteres en el orden correcto, por la naturaleza de la variable acumuladora, los nuevos caracteres se guardan en las primeras posiciones del acumulador, siendo necesario darles la vuelta con el predicado **`reverse/2`**.

Si lo que teníamos en el acumulador esta vació eso significa que solo vamos a necesitar añadir el carácter leído a nuestra lista de tokens, en caso de que en el acumulador teníamos algún carácter, se concatenan estos caracteres para formar la palabra completa (un átomo) con la ayuda del predicado definido **`atomic_list_concat/2`**. Cabe la posibilidad de que ese átomo creado pueda ser un número, ya que en el futuro los números y los átomos van a tener comportamientos distintos, es necesario comprobarlo con el predicado definido **`atom_number/2`** y vemos si es posible realizar la conversión o no, añadiendo o el número correspondiente o en su contrario el átomo.

```
%Caso de no encontrar un espacio,tabulador, en general delimitadores de espacio: se acumula
concatenar_caracteres_al_reves([Caracter|Caracteres],Acumulador,Tokens) :-
    % Si en el acumulador tenemos el '\'' eso significa que aceptamos cualquier tipo de cadena
    ( member('\'',Acumulador) ->
        concatenar_caracteres_al_reves(Caracteres,[Caracter|Acumulador],Tokens)
        ;
        \+member(Caracter,[' ','\n','\t']),!,concatenar_caracteres_al_reves(Caracteres,[Caracter|Acumulador],Tokens)
    ).
```

Otro caso que encontramos es el escenario de leer un carácter que no sea especial y añadirlo en el acumulador. Como se puede apreciar, cabe resaltar si encontramos el carácter '\'' que corresponde a una comilla simple, todo lo que vaya después de leer dicho carácter, lo vamos a almacenar porque estamos hablando de una cadena de caracteres de tipo String, por lo que también deberíamos aceptar los delimitadores espaciales como el espacio, el tabulador o el salto de línea.

```
%Caso de encontrar un espacio: se junta los caracteres (no tiene que ser vacio el acum) y se mete en la lista Tokens
concatenar_caracteres_al_reves([_|Caracteres],Acumulador,[Token|Tokens]):-
    reverse(Acumulador, Reversa),
    ( Acumulador \= [] ->
        atomic_list_concat(Reversa, Atomo),
        ( atom_number(Atomo,Numero) ->
            Token = Numero
            ;
            Token = Atomo
        )
    ),
    concatenar_caracteres_al_reves(Caracteres,[],Tokens).
```

Encontrando con uno de estos delimitadores espaciales mencionados, pero sin que le precedan el símbolo de la comilla simple, se habla de un delimitador, por lo que procedemos a invertir el acumulador para tener la secuencia en orden, concatenar la secuencia de caracteres y comprobar si es un numero o un átomo.

```
%Caso especifica donde exite 1 caracter que debemos omitir
concatenar_caracteres_al_reves([_|Caracteres],_,Tokens) :- concatenar_caracteres_al_reves(Caracteres,[],Tokens).
```

Como último caso tenemos si no hemos metido nada en el acumulador y nos topamos con un carácter que actúa como delimitador espacial, que el programa simplemente no lo tome en cuenta y sigua leyendo los caracteres que le falten.

2.2.- Parser

Como segunda fase de la compilación, se construirá el árbol semántico para que se pueda interpretar y computar por la maquina abstracta. Para ello, vamos a utilizar gramáticas de cláusulas definidas que nos proporciona una sintaxis sencilla para poder crear el árbol semántico con sus componentes utilizando el predicado definido **phrase/2**, donde el primer argumento corresponde a la definición de la gramática causal, que devolverá el árbol semántico y como segundo parámetro la lista de tokens.

```
pl_program_parser(S) --> [program], identifier(_),[';'],statement(S).

statement((S;Ss);S3) --> [var], statement_var(S), rest_var(Ss), statement(S3). %Multiples variables
statement((S;S3)) --> [var], statement_var(S), statement(S3). % Unica variable
```

En una idea general, un programa va a estar formado por la cadena **program**, un identificador y un cuerpo del programa a ejecutar, siendo este cuerpo las diferentes instrucciones. También definimos una sentencia cuando vamos a declarar variables al principio del programa, por eso estas reglas vienen definidas al principio con la cadena **var**, seguido de o bien de varias declaraciones de variables o de una sola.

```
statement_var(var(X)) --> identifier(X).
rest_var((S;Ss)) --> [';'], statement_var(S), rest_var(Ss).
rest_var(void) --> [';'].
```

Una variable viene definida como var(X) y esto será cierto si X es un identificador que viene dada por la regla **`identifier(name(X)) --> [X],{atom(X)}`**, es decir, X será un identificador si X es un átomo. Para poder definir más declaraciones de variables se creó la regla de **`rest_var`**, para definir más definición de variables o en caso contrario parar la llamada recursiva con la segunda definición de **`rest_var(void)`** ya que toda declaración de variables termina con el ';' final de la última variable.

```
statement((S;Ss)) --> [begin] , statement(S), rest_statements(Ss).
```

Una vez definidas las variables, pasaríamos al propio cuerpo de las instrucciones de nuestro programa, que se corresponde a la secuencia de una instrucción y el resto de las instrucciones, formándose el árbol semántico recursivamente. Ahora habría que mirar que tipo de instrucción hay que definir y cómo se define el resto de las instrucciones.

```
rest_statements((S;Ss)) --> [';'] , statement(S), rest_statements(Ss).
rest_statements(void) --> [';'], [end], ['.'].
rest_statements(void) --> [';'] , [end].
rest_statements(void) --> [';'] , [end], [';'].
rest_statements(void) --> [end],['.'].
rest_statements(void) --> [end].
```

La primera definición de **`rest_statements((S;Ss))`** correspondería al caso normal recursivo, donde tenemos una instrucción después de otra, así hasta llegar a un caso base, donde nos encontraremos la cadena **end** en sus diferentes casuísticas.

A continuación, vamos a ver los diferentes tipos de instrucciones que podría tener un programa dentro de la regla de **`statement`**:

- **Asignación**

```
statement(assign(X,V)) --> identifier(X) ,[':'],['='], expression(V).
```

La definición de asignación viene correspondida con guardar en la variable/identificador X una expresión V, precedido por el símbolo de asignación **':='.**

```
expression(X) --> pl_constant(X).
expression(expr(Op,X,Y)) --> pl_constant(X), arithmetic_op(Op), expression(Y).
arithmetic_op('+') --> ['+'].
arithmetic_op('-') --> ['-'].
arithmetic_op('*') --> ['*'].
arithmetic_op('/') --> ['/'].
```

Una expresión viene definida de una constante, seguido de un operador operacional y otra expresión, por lo que podríamos tener una expresión tan larga como queramos.

Una constante puede ser o bien un numero entero o un identificador, por lo que a esa variable tendrá asociado un numero entero para poder realizar la operación.

- **Condicionales**

```
statement(if(T,S1,S2)) --> [if],['('],test(T),[')'],[then] , statement(S1), [else] , statement(S2).
statement(if(T,S1)) --> [if],['('],test(T),[')'],[then],statement(S1).
```

Dentro de las condicionales, tenemos 2 estructuras, el **if-then-else,** en caso de querer ejecutar instrucciones si no se da la condición, y tenemos la estructura **if-then,** ambas seguidas de 1 o varias instrucciones.

```
test(compare(Op,X,Y)) --> expression(X), comparison_op(Op), expression(Y).
comparison_op('=') --> ['='].
comparison_op('<') --> ['<'].
comparison_op('>') --> ['>'].
comparison_op('>=') --> ['>'],['='].
comparison_op('<=') --> ['<'],['='].
```

Para el concepto de comprobar 2 expresiones, definidas en la regla **test** hay que tener en cuenta que lo que se vaya a evaluar son 2 expresiones utilizando un operador lógico.

- **While**

```
statement(while(T,S)) --> [while],['('],test(T),[')'],[do],statement(S).
```

Un bucle while viene definida por los caracteres propios de un bucle while que es el propio "while" y el "do" seguido de las instrucciones que se tiene que ejecutar en el bucle con su respectiva comprobación del bucle con **test(T).**

- **For**

```
statement(for(T,S1,S2)) --> [for], expression(T), [to],expression(S1), [do], statement(S2).
statement(for(T,S1,S2)) --> [for], statement(T), [to],expression(S1), [do], statement(S2).
```

Tenemos 2 definiciones de bucle for que corresponden a la siguiente estructura:

- **Primera forma**

'for' identificador ':=' entero 'to' entero 'do'

begin

instrucciones

end;

- **Segunda forma**

'for' identificador 'to' entero 'do'

begin

instrucciones

end;

- **Read**

```
statement(read(X)) --> [read] ,['('],identifier(X),[')'].
```

Esta definición nos indica que vamos a poder atribuir un valor a las variables.

- **Write/Writeln**

```
statement(write(X)) --> [write] ,['('],expression(X),[')'].
statement(writeln(X)) --> [writeln] ,['('],expression(X),[')'].
```

Podemos imprimir por pantalla con esta definición expresiones, que o pueden ser constantes o en su defecto una expresión aritmética.

2.3.- Maquina Abstracta

La máquina abstracta en Prolog es esencial para la ejecución de programas, ya que se encarga de interpretar y ejecutar las instrucciones generadas a partir del análisis sintáctico del programa fuente.

Este proceso se divide en dos etapas:

- La construcción de la lista de instrucciones
- La ejecución de dichas instrucciones

En esta fase, una vez que se ha realizado el análisis léxico y sintáctico del programa fuente, se obtiene una cadena de tokens que representa las instrucciones del programa en un formato comprensible para la máquina abstracta. Esta cadena de tokens se pasa al compilador de Prolog, donde se procede a la construcción de la lista de instrucciones. Esta lista es esencial para la ejecución del programa, ya que contiene todas las acciones que la máquina abstracta debe realizar para ejecutar el programa de manera adecuada.

Para la construcción de la lista hay 2 funciones:

- parse_list:

```
parse_list(Estructura,List) :- compound(Estructura),compound_name_arguments(Estructura, ';', Arguments),
!, split_compound(Arguments,List).

parse_list(X,[X]) :- not(is_list(X)).
```

Esta función está formada por la regla base en la que, si X no es una lista en este caso, simplemente se devuelve X como unica instrucción de la lista y la regla recursiva, en este caso si “Estructura” es un arbol formado por varios elementos, se utiliza la función compound_name_arguments, con la que se obtienen los argumentos y se selecciona como funtor “;”. Por último, split_compound coge los argumentos obtenidos anteriormente y y la divide en una lista de elementos.

- Split_compound:

```
split_compound([],[]).
split_compound([T|Ts],L) :- parse_list(T,L1),split_compound(Ts,L2),append(L1,L2,L).
```

Esta función está formada por la regla base que, en caso de tener una lista vacía, devuelve la lista vacía y luego el caso recursivo en el que se toma el primer elemento T de la lista y se procesa para obtener su lista de instrucciones “L1”, luego se llama recursivamente con el resto de la lista “Ts” para obtener la lista de instrucciones restante “L2”. Finalmente, se concatenan “L1” y “L2” para obtener la lista completa de instrucciones “L”.

Una vez construida la lista de instrucciones, la máquina abstracta entra en la fase de ejecución, donde se lleva a cabo la interpretación y ejecución de cada instrucción de manera secuencial. Durante este proceso, la máquina abstracta utiliza un diccionario de variables para llevar un registro del estado actual del programa y actualizarlo según sea necesario. Cada instrucción se evalúa

individualmente, y se realizan las acciones correspondientes, como asignación de valores a variables, operaciones aritméticas, lectura y escritura de datos, y control de flujo del programa.

Para la ejecución de las instrucciones:

El predicado principal es computar_instr :

```
computar_instr(Dicc,[],Dicc).A
computar_instr(Dicc,[I|Is],DiccF) :- eval(Dicc,I,Dicc2),nl,nl,computar_instr(Dicc2,Is,DiccF).
```

Formada por el caso base que indica que cuando la lista de instrucciones está vacía, el diccionario de estado final (Dicc) es igual al diccionario de estado actual (Dicc) y el caso recursivo en el que “I” representa la instrucción inicial y “Is” el resto de las instrucciones.

El evaluador tiene como predicado principal “eval(Dict,Termino,Dict1), siendo Dict el diccionario antes de la evaluación y Dict1 el diccionario con el estado tras la evaluación; Término será el item parseado a evaluar, que puede tomar muchas formas por ello tendremos diversas reglas para poder controlarlo:

```
eval(Dicc,write(name(X)),Dicc) :-
    (es_cadena_char(X) ->
        write(X)
        ;
        get_dict(X,Dicc,Val),write(Val)
    ).
eval(Dicc,writeln(name(X)),Dicc) :-
    (es_cadena_char(X) ->
        writeln(X)
        ;
        get_dict(X,Dicc,Val),writeln(Val)
    ).
```

Ambos predicados se encargan de imprimir en pantalla el valor contenido en la variable X. X puede ser tanto una cadena de caracteres como el nombre de una variable.

Si X es una cadena de caracteres, se imprime directamente usando write(X) o writeln(X). Si X es el nombre de una variable, se obtiene su valor del diccionario Dicc usando get_dict(X, Dicc, Val) y se imprime ese valor.

```
eval(Dicc,var(name(X)),DiccF) :- DiccF=Dicc.put(X,0).
```

Este predicado se utiliza para declarar una nueva variable con nombre X en el diccionario de estado Dicc. La variable recién creada se inicializa con el valor 0.

```
eval(Dicc,read(name(X)),DiccF) :- read(N),DiccF=Dicc.put(X,N).
```

Esta cláusula se utiliza para leer un valor de entrada del usuario y asignarlo a la variable X. El valor leído se almacena en la variable N. Luego, se actualiza el diccionario de estado Dicc para asociar el valor N con la variable X usando DiccF=Dicc.put(X,N).

```
eval(Dicc,assign(name(X),number(N)),DiccF) :- DiccF=Dicc.put(X,N).
eval(Dicc,assign(name(X),name(Y)),DiccF) :- get_dict(Y, Dicc, N), DiccF=Dicc.put(X,N).
eval(Dicc,assign(name(X),Expr),DiccF) :- calcular_expr(Expr,Dicc,N), DiccF=Dicc.put(X,N).
```

Estos tres predicados se utilizan para asignar un valor a la variable X. Dependiendo del tipo de valor que se esté asignando, se manejan diferentes situaciones:

- Si el valor es un número, se asigna directamente.
- Si el valor es el nombre de otra variable (name(Y)), se obtiene el valor de Y del diccionario Dicc.
- Si el valor es una expresión (Expr), se calcula su valor utilizando el predicado calcular_expr y se asigna el resultado a X.

```
eval(Dicc,if(Cond,Acc),DiccF) :- (comparar_aux(Cond,Dicc) -> computar(Dicc,Acc,DiccF);DiccF=Dicc),!. % if sin else
eval(Dicc,if(Cond,Acc1,Acc2),DiccF) :- (comparar_aux(Cond,Dicc) -> computar(Dicc,Acc1,DiccF);computar(Dicc,Acc2,DiccF)),!. % if else
```

Estos dos predicados implementan la estructura de control condicional "if". Cond es la condición que se evalúa.

Tenemos por un lado el predicado que se ejecuta en el caso de solo tener un if y el que se ejecuta en el caso de que el if tenga else, en este caso si se cumplen la primera condición se cumple la Acc1 y si no la Acc2.

```
eval(Dicc,while(Cond,Ins),DiccF) :- comparar_aux(Cond,Dicc),computar(Dicc,Ins,Dicc2),eval(Dicc2,while(Cond,Ins),DiccF);DiccF=Dicc.
```

En este predicado si la condición se cumple, se ejecuta el código. Una vez ejecutado se vuelve a evaluar nuevamente la condición, si se cumple se llama nuevamente a este predicado, pero con los valores del Dicc actualizados en caso de que no se cumpla se devuelve el estado actual del diccionario como diccionario final.

```
eval(Dicc,for(name(I),name(X),Ins),DiccF) :-
    (comparar_aux(compare(<,name(I),name(X)),Dicc)
    -> computar(Dicc,Ins,D2),eval(D2,assign(name(I),expr(+,name(I),number(1))),D3),eval(D3,for(name(I),name(X),Ins),DiccF)
    ;
    DiccF = Dicc
    ).

eval(Dicc,for(assign(name(I),number(N)),name(X),Ins),DiccF) :-
    eval(Dicc,assign(name(I),number(N)),D2),
    (comparar_aux(compare(<,name(I),name(X)),D2)
    -> computar(D2,Ins,D3),eval(D3,assign(name(I),expr(+,name(I),number(1))),D4),eval(D4,for(name(I),name(X),Ins),DiccF)
    ;
    DiccF = Dicc
    ).
```

Estos dos predicados sirven para los dos casos del for:

Caso 1: for(name(I), name(X), Ins): Aquí, se verifica si el valor de la variable I es menor que el valor de la variable X en el diccionario Dicc. Si es así, se ejecutan las instrucciones Ins, luego se incrementa el valor de I en 1, y se llama recursivamente a eval para la próxima iteración. Este proceso continúa hasta que la condición I < X ya no se cumple. Si I ya es mayor o igual a X al inicio, las instrucciones no se ejecutan y se devuelve el mismo diccionario Dicc.

Caso 2: for(assign(name(I), number(N)), name(X), Ins): En este caso, primero se evalúa la asignación de un valor inicial N a la variable I. Luego, se verifica si el valor de I es menor que el valor de X en el diccionario Dicc. Si es así, se ejecutan las instrucciones Ins, luego se incrementa el valor de I en 1, y se llama recursivamente a eval para la próxima iteración. Este proceso continúa hasta que la condición I < X ya no se cumple. Si I ya es mayor o igual a X al inicio, las instrucciones no se ejecutan y se devuelve el mismo diccionario Dicc.

```
eval(Dicc,void,Dicc).
```

Sirve para representar el final de una secuencia del parser, ya sea el final como tal de todos los términos, el final de un bucle, de una secuencia condicional ya que se envía un void al final de cada una de estas secuencias.

Para poder utilizar de manera correcta estos predicados es necesario crear otros auxiliares que son:

```
calcular_expr(expr(Op,number(X),number(Y)),_,N) :- calcular(Op,X,Y,N).
calcular_expr(expr(Op,name(X),number(Y)),Dicc,N) :- get_dict(X,Dicc,Val),calcular(Op,Val,Y,N).
calcular_expr(expr(Op,number(X),name(Y)),Dicc,N) :- get_dict(Y,Dicc,Val),calcular(Op,X,Val,N).
calcular_expr(expr(Op,name(X),name(Y)),Dicc,N) :- get_dict(X,Dicc,ValX),get_dict(Y,Dicc,ValY),calcular(Op,ValX,ValY,N).
calcular_expr(expr(Op,name(X),Expr),Dicc,N) :- calcular_expr(Expr,Dicc,N1),get_dict(X,Dicc,Val),calcular(Op,Val,N1,N).
calcular_expr(expr(Op,number(X),Expr),Dicc,N) :- calcular_expr(Expr,Dicc,N1), calcular(Op,X,N1,N).

calcular(+,X,Y,N) :- N is X + Y.
calcular(-,X,Y,N) :- N is X - Y.
calcular(*,X,Y,N) :- N is X * Y.
calcular(/,X,Y,N) :- N is X / Y.
```

Primero contamos con los de calcular una expresión en la que hay que considerar que hay 4 operaciones [+, -, *, /] y también todas las combinaciones posibles, número con número, número con variable, variable con número, variable con variable.

En las dos últimas hay que tener en cuenta que, si uno de los dos valores es una expresión, hay que realizar esas operaciones antes de calcular la operación final

```
comparar_aux(compare(OpL,number(X),number(Y)),_) :- comparar(OpL,X,Y),!.
comparar_aux(compare(OpL,number(X),name(Y)),Dicc) :- get_dict(Y,Dicc,Val), comparar(OpL,X,Val),!.
comparar_aux(compare(OpL,name(X),number(Y)),Dicc) :- get_dict(X,Dicc,Val),comparar(OpL,Val,Y),!.
comparar_aux(compare(OpL,name(X),name(Y)),Dicc) :- get_dict(X,Dicc,ValX),get_dict(Y,Dicc,ValY), comparar(OpL,ValX,ValY),!.

comparar(=,X,Y) :- X =:= Y.
comparar(<,X,Y) :- X < Y.
comparar(>,X,Y) :- X > Y.
comparar(<=,X,Y) :- X =< Y.
comparar(>=,X,Y) :- X >= Y.
```

Luego tenemos las comparaciones que funcionan de la misma manera que las operaciones, debemos tener en cuenta los tipos de comparadores [=, <, >, <=, =>], además de tener en cuenta las combinaciones de número con números, número con variable, variable con número, variable con variable.

```
es_cadena_char(Atomo) :- atom_chars(Atomo, [Primero|_]),Primero == '\''.
```

Por último, tenemos este predicado verifica si un átomo comienza con una comilla simple, lo que nos dice que representa una cadena de caracteres.

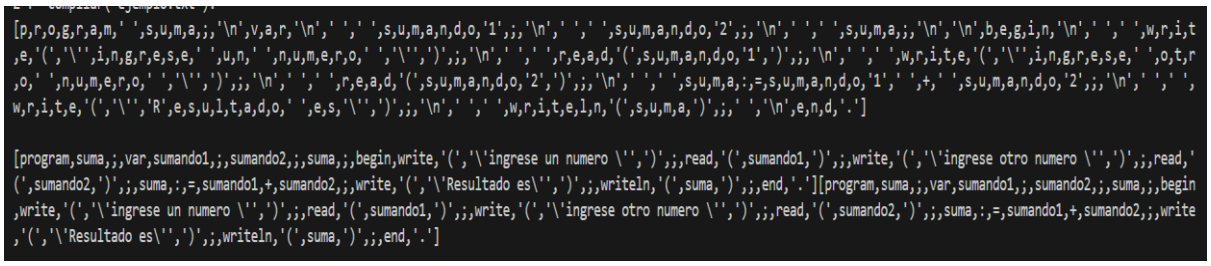
2.4.- Ejemplo de ejecución

Una vez explicado el funcionamiento del programa, enseñamos algunos ejemplos de programas típicos de programación.

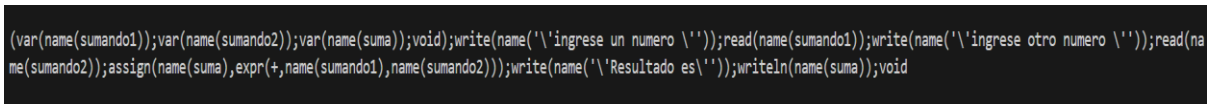
Suma

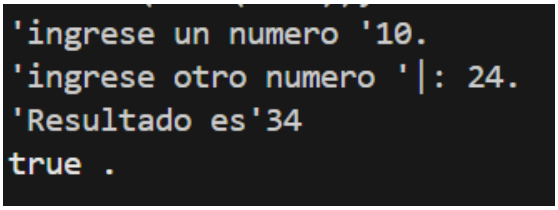
```
program suma;
var
  sumando1;
  sumando2;
  suma;

begin
  write('ingrese un numero ');
  read(sumando1);
  write('ingrese otro numero ');
  read(sumando2);
  suma:=sumando1 + sumando2;
  write('Resultado es');
  writeln(suma);
end.
```

Como podemos ver lo primero que hace el programa es dividir en tokens:



El siguiente paso es usar el paser para pasarlo a la gramática DCG:





Introducimos las variables que nos piden y obtenemos la solución final.

Factorial

```
program factorial;
var
value;count;result;
begin
read (value);
count := 1;
result := 1;
while (count < value) do
begin
count:= count+1;
result := result*count;
end;
writeln (result);
end.
```

Como podemos ver lo primero que hace el programa es dividir en tokens:

```
1 ?- compilar("factorial.txt").
[p,r,o,g,r,a,m,' ',f,a,c,t,o,r,i,a,l,;,'\n',v,a,r,'\n',v,a,l,u,e,;,c,o,u,n,t,;,r,e,s,u,l,t,;,'\n',b,e,g,i,n,'\n',r,e,a,d,' ','(',v,a,l,u,e,')',;,'\n',c,o,u,n,t,' ',:,=,'
 ','1',;,' ','\n',r,e,s,u,l,t,' ',:,=,' ','1',;,' ','\n',w,h,i,l,e,' ','(',c,o,u,n,t,' ',<,' ',v,a,l,u,e,')',' ',d,o,'\n',b,e,g,i,n,'\n',c,o,u,n,t,:,=,' ',c,o,u,n,t,+,'1
',;,'\n',r,e,s,u,l,t,' ',:,=,' ',r,e,s,u,l,t,*,c,o,u,n,t,;,'\n',e,n,d,;,'\n',w,r,i,t,e,l,n,' ','(',r,e,s,u,l,t,')',;,'\n',e,n,d,'.']
```

```
[program,factorial,;,var,value,;,count,;,result,;,begin,read,'(',value,')',;,count,:,=,1,;,result,:,=,1,;,while,'(',count,<,value,')',do,begin,count,:,=,count,+,1,;,resu
lt,:,=,result,*,count,;,end,;,writeln,'(',result,')',;,end,'.'][program,factorial,;,var,value,;,count,;,result,;,begin,read,'(',value,')',;,count,:,=,1,;,result,:,=,1,;,
while,'(',count,<,value,')',do,begin,count,:,=,count,+,1,;,result,:,=,result,*,count,;,end,;,writeln,'(',result,')',;,end,'.']
```

El siguiente paso es usar el paser para pasarlo a la gramática DCG:

```
(var(name(value));var(name(count));var(name(result));void);read(name(value));assign(name(count),number(1));assign(name(result),number(1));while(compare(<,name(count),nam
e(value)),(assign(name(count),expr(+,name(count),number(1)));assign(name(result),expr(*,name(result),name(count)));void));writeln(name(result));void
```

```
|: 5.
120
true .
```

Resultado esperado de realizar el factorial de 5

Fibonacci

```
program fibonacci;
var
  i;
  n;
  a;
  b;
  c;

begin
  a:=0;
  b:=1;
  read(n);
  if (n = 0) then
  begin
    writeln(0);
  end;
  if (n = 1) then
  begin
    writeln(1);
  end;
  if (n > 1) then
  begin
    for i := 1 to n do
    begin
      c := a + b;
      a := b;
      b := c;
    end;
    writeln(b);
  end;
end.
```

Como podemos ver lo primero que hace el programa es dividir en tokens:

```
3 ?- compilar("fibonacci.txt").
[p,r,o,g,r,a,m,' ',f,i,b,o,n,a,c,c,i,;,;,'\n',v,a,r,'\n',' ',' ',i,;,;,'\n',' ',' ',n,;,;,'\n',' ',' ',a,;,;,'\n',' ',' ',b,;,;,'\n',' ',' ',c,;,;,'\n','\n',b,e,g,i,n,'\n',' ',' ',a
,:,=,'0',;,;,'\n',' ',' ',b,:,=,'1',;,;,'\n',' ',' ',r,e,a,d,'(',n,')',;,;,'\n',' ',' ',i,f,' ','(',n,' ',=,' ','0',')',' ',t,h,e,n,'\n',' ',' ',b,e,g,i,n,'\n',' ',' ',' ',' ',
,w,r,i,t,e,l,n,'(','0',')',;,;,'\n',' ',' ',e,n,d,;,;,'\n',' ',' ',i,f,' ','(',n,' ',=,' ','1',')',' ',t,h,e,n,'\n',' ',' ',b,e,g,i,n,'\n',' ',' ',' ',' ',w,r,i,t,e,l,n,'(',
'1',')',;,;,'\n',' ',' ',e,n,d,;,;,'\n',' ',' ',i,f,' ','(',n,' ','>',' ','1',')',' ',t,h,e,n,'\n',' ',' ',b,e,g,i,n,'\n',' ',' ',' ',' ',f,o,r,' ',i,' ',:,=,' ','1',' ',t,o,'
 ',n,' ',d,o,'\n',' ',' ',' ',' ',b,e,g,i,n,'\n',' ',' ',' ',' ',' ',' ',' ',c,' ',:,=,' ',a,' ',+,' ',b,;,;,'\n',' ',' ',' ',' ',' ',' ',' ',a,' ',:,=,' ',b,;,;,'\n',' ',' ',' ',
' ',' ',' ',b,' ',:,=,' ',c,;,;,'\n',' ',' ',' ',' ',' ',e,n,d,;,;,'\n',' ',' ',' ',' ',w,r,i,t,e,l,n,'(',b,')',;,;,'\n',' ',' ',e,n,d,;,;,'\n',e,n,d,'.']

[program,fibonacci,;,;,var,i,;,;,n,;,;,a,;,;,b,;,;,c,;,;,begin,a,:,=,0,;,;,b,:,=,1,;,;,read,'(',n,')',;,;,if,'(',n,=,0,')',then,begin,writeln,'(',0,')',;,;,end,;,;,if,'(',n,=,1,')',then,begin
,writeln,'(',1,')',;,;,end,;,;,if,'(',n,>,1,')',then,begin,for,i,:,=,1,to,n,do,begin,c,:,=,a,+,b,;,;,a,:,=,b,;,;,b,:,=,c,;,;,end,;,;,writeln,'(',b,')',;,;,end,;,;,end,'.'][program,fibon
acci,;,;,var,i,;,;,n,;,;,a,;,;,b,;,;,c,;,;,begin,a,:,=,0,;,;,b,:,=,1,;,;,read,'(',n,')',;,;,if,'(',n,=,0,')',then,begin,writeln,'(',0,')',;,;,end,;,;,if,'(',n,=,1,')',then,begin,writeln,'(',1
,')',;,;,end,;,;,if,'(',n,>,1,')',then,begin,for,i,:,=,1,to,n,do,begin,c,:,=,a,+,b,;,;,a,:,=,b,;,;,b,:,=,c,;,;,end,;,;,writeln,'(',b,')',;,;,end,;,;,end,'.']
```

El siguiente paso es usar el paser para pasarlo a la gramática DCG:

```
(var(name(i));var(name(n));var(name(a));var(name(b));var(name(c));void);assign(name(a),number(0));assign(name(b),number(1));read(name(n));if(compare(=,name(n),number(0))
,(writeln(number(0));void));if(compare(=,name(n),number(1)),(writeln(number(1));void));if(compare(>,name(n),number(1)),(for(assign(name(i),number(1)),name(n),(assign(nam
e(c),expr(+,name(a),name(b)));assign(name(a),name(b));assign(name(b),name(c));void));writeln(name(b));void));void
```

```
|: 10.
55
true .
```

Resultado esperado al realizar el Fibonacci de 10

3.- Código fuente

```prolog
%%%%%%%%%%%%%%%%%%%%%%% PREDICADOS CORRESPONDIENTE AL FUNCIONAMIENTO DEL COMPILADOR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

compilar(Fichero) :- tokenizar(Tokens,Fichero),parse(Tokens,Estructura),computar(dict{},Estructura,_).

tokenizar(Tokens,Fichero) :- obtener_listado_car(Fichero,Lista_char),print(Lista_char),procesar_tokens(Lista_char,Tokens),print(Tokens),!,nl.
parse(Source,Estructura) :- phrase(pl_program_parser(Estructura),Source),!,nl,print(Estructura),nl.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Predicado para leer un archivo caracter a caracter
leer_caracteres(Stream, Lista) :-
    at_end_of_stream(Stream), !, % Verificar si hemos llegado al final del archivo
    Lista = []. % Si hemos llegado al final, la lista está vacía

leer_caracteres(Stream, [Caracter|Cola]) :-
    get_char(Stream, Caracter), % Leer el siguiente carácter como un atomo
    leer_caracteres(Stream, Cola). % Leer el resto de los caracteres


obtener_listado_car(Fichero,Lista_char) :- open(Fichero, read, Stream),
   leer_caracteres(Stream,Lista_char),
   close(Stream).
```

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PASO 1: CREACIÓN DEL TOKENISER %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

procesar_tokens(Caracteres,Tokens) :- concatenar_caracteres_al_reves(Caracteres,[],Tokens),nl,nl,print(Tokens).

concatenar_caracteres_al_reves([],[],[]) :- !.

%Caso base: ultimo token
concatenar_caracteres_al_reves([],Acumulador,[Token|Tokens]) :-
    reverse(Acumulador, Reversa), atomic_list_concat(Reversa, Token), concatenar_caracteres_al_reves([],[],Tokens).

%------------------------Caso general/es-----------------------------------

%Caso de ser un caracter especial (; , ( ,),...) monocaracter
concatenar_caracteres_al_reves([Caracter|Caracteres], Acumulador, Tokens) :-
    member(Caracter, [';',':','=','(', ')','+','-','*','/',',','<','>','.']), !,
    reverse(Acumulador, Reversa),
    ( Reversa \= [] ->
        atomic_list_concat(Reversa, Atomo),
        ( atom_number(Atomo, Numero) ->
            Tokens = [Numero,Caracter|RestoTokens]
            ;
            Tokens = [Atomo,Caracter|RestoTokens]
        )
        ;
        Tokens = [Caracter|RestoTokens]
    ),
    concatenar_caracteres_al_reves(Caracteres, [], RestoTokens).
```

```prolog
%Caso de no encontrar un espacio,tabulador, en general delimitadores de espacio: se acumula
concatenar_caracteres_al_reves([Caracter|Caracteres],Acumulador,Tokens) :-
    % Si en el acumulador tenemos el '\'' eso significa que aceptamos cualquier tipo de cadena
    ( member('\'',Acumulador) ->
        concatenar_caracteres_al_reves(Caracteres,[Caracter|Acumulador],Tokens)
        ;
        \+member(Caracter,[' ','\n','\t']),!,concatenar_caracteres_al_reves(Caracteres,[Caracter|Acumulador],Tokens)
    ).

%Caso de encontrar un espacio: se junta los caracteres (no tiene que ser vacio el acum) y se mete en la lista Tokens
concatenar_caracteres_al_reves([_|Caracteres],Acumulador,[Token|Tokens]):-
    reverse(Acumulador, Reversa),
    ( Acumulador \= [] ->
        atomic_list_concat(Reversa, Atomo),
        ( atom_number(Atomo,Numero) ->
            Token = Numero
            ;
            Token = Atomo
        )
    ),
    concatenar_caracteres_al_reves(Caracteres,[],Tokens).

%Caso especifica donde exite 1 caracter que debemos omitir
concatenar_caracteres_al_reves([_|Caracteres],_,Tokens) :- concatenar_caracteres_al_reves(Caracteres,[],Tokens).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PASO 2: CREACIÓN DEL PARSER %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% parse(Tokens,Estructura) donde Estructura representa la lista de tokens parseada correctamente

pl_program_parser(S) --> [program], identifier(_),[';'],statement(S).

statement((S;Ss);S3) --> [var], statement_var(S), rest_var(Ss), statement(S3). %Multiples variables
statement((S;S3)) --> [var], statement_var(S), statement(S3). % Unica variable

statement((S;Ss)) --> [begin] , statement(S), rest_statements(Ss).
statement(assign(X,V)) --> identifier(X) ,[':'],['='], expression(V).
statement(if(T,S1,S2)) --> [if],['('],test(T),[')'],[then] , statement(S1), [else] , statement(S2).
statement(if(T,S1)) --> [if],['('],test(T),[')'],[then],statement(S1).
statement(while(T,S)) --> [while],['('],test(T),[')'],[do],statement(S).
statement(for(T,S1,S2)) --> [for], expression(T), [to],expression(S1), [do], statement(S2).
statement(for(T,S1,S2)) --> [for], statement(T), [to],expression(S1), [do], statement(S2).

statement(read(X)) --> [read] ,['('],identifier(X),[')'].
statement(write(X)) --> [write] ,['('],expression(X),[')'].
statement(writeln(X)) --> [writeln] ,['('],expression(X),[')'].

statement_var(var(X)) --> identifier(X).
rest_var((S;Ss)) --> [';'], statement_var(S), rest_var(Ss).
rest_var(void) --> [';'].

rest_statements((S;Ss)) --> [';'] , statement(S), rest_statements(Ss).
rest_statements(void) --> [';'], [end], ['.'].
rest_statements(void) --> [';'] , [end].
rest_statements(void) --> [';'] , [end], [';'].
rest_statements(void) --> [end],['.'].
rest_statements(void) --> [end].

expression(X) --> pl_constant(X).
expression(expr(Op,X,Y)) --> pl_constant(X), arithmetic_op(Op), expression(Y).
arithmetic_op('+') --> ['+'].
arithmetic_op('-') --> ['-'].
arithmetic_op('*') --> ['*'].
arithmetic_op('/') --> ['/'].
```

```
pl_constant(X) --> pl_integer(X).
pl_constant(X) --> identifier(X).
pl_integer(number(X)) --> [X],{integer(X)}.
identifier(name(X)) --> [X], {atom(X)}.

test(compare(Op,X,Y)) --> expression(X), comparison_op(Op), expression(Y).
comparison_op('=') --> ['='].
comparison_op('<') --> ['<'].
comparison_op('>') --> ['>'].
comparison_op('>=') --> ['>'],['='].
comparison_op('<=') --> ['<'],['='].
```

```prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PASO 3: MÁQUINA ABSTRACTA %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Importamos la librería para la gestión del estado de la maquina
:- use_module(library(dicts)).

% Llamada principal para el computo de instrucciones del programa
computar(Dicc,Estructura,DiccF) :- parse_list(Estructura,L),computar_instr(Dicc,L,DiccF).

% 3.1.- Construir la lista de instrucciones a partir del árbol
parse_list(Estructura,List) :- compound(Estructura),compound_name_arguments(Estructura, ';', Arguments),
!, split_compound(Arguments,List).

parse_list(X,[X]) :- not(is_list(X)).

split_compound([],[]).
split_compound([T|Ts],L) :- parse_list(T,L1),split_compound(Ts,L2),append(L1,L2,L).

% 3.2.- Ir ejecutando instruccion por instruccion
computar_instr(Dicc,[],Dicc).
computar_instr(Dicc,[I|Is],DiccF) :- eval(Dicc,I,Dicc2),nl,print(Dicc),nl,computar_instr(Dicc2,Is,DiccF).

%get_dict(X,DiccF,Value),writeln(Value).

% Proceso de evaluación de instrucciones
eval(Dicc,void,Dicc).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
eval(Dicc,write(name(X)),Dicc) :-
    (es_cadena_char(X) ->
        write(X)
        ;
        get_dict(X,Dicc,Val),write(Val)
    ).
eval(Dicc,writeln(name(X)),Dicc) :-
    (es_cadena_char(X) ->
        writeln(X)
        ;
        get_dict(X,Dicc,Val),writeln(Val)
    ).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
eval(Dicc,var(name(X)),DiccF) :- DiccF=Dicc.put(X,0). %Declaracion de variable
eval(Dicc,read(name(X)),DiccF) :- read(N),DiccF=Dicc.put(X,N).

eval(Dicc,assign(name(X),number(N)),DiccF) :- DiccF=Dicc.put(X,N).
eval(Dicc,assign(name(X),name(Y)),DiccF) :- get_dict(Y, Dicc, N), DiccF=Dicc.put(X,N).
eval(Dicc,assign(name(X),Expr),DiccF) :- calcular_expr(Expr,Dicc,N), DiccF=Dicc.put(X,N).

eval(Dicc,if(Cond,Acc),DiccF) :- (comparar_aux(Cond,Dicc) -> computar(Dicc,Acc,DiccF);DiccF=Dicc),!. % if sin else
eval(Dicc,if(Cond,Acc1,Acc2),DiccF) :- (comparar_aux(Cond,Dicc) -> computar(Dicc,Acc1,DiccF);computar(Dicc,Acc2,DiccF)),!. % if else

eval(Dicc,while(Cond,Ins),DiccF) :- comparar_aux(Cond,Dicc),computar(Dicc,Ins,Dicc2),eval(Dicc2,while(Cond,Ins),DiccF);DiccF=Dicc.

eval(Dicc,for(name(I),name(X),Ins),DiccF) :-
    (comparar_aux(compare(<,name(I),name(X)),Dicc)
    -> computar(Dicc,Ins,D2),eval(D2,assign(name(I),expr(+,name(I),number(1))),D3),eval(D3,for(name(I),name(X),Ins),DiccF)
    ;
    DiccF = Dicc
    ).

eval(Dicc,for(assign(name(I),number(N)),name(X),Ins),DiccF) :-
    eval(Dicc,assign(name(I),number(N)),D2),
    (comparar_aux(compare(<,name(I),name(X)),D2)
    -> computar(D2,Ins,D3),eval(D3,assign(name(I),expr(+,name(I),number(1))),D4),eval(D4,for(name(I),name(X),Ins),DiccF)
    ;
    DiccF = Dicc
    ).
```

```
 %Metodos auxiliares
 calcular_expr(expr(Op,number(X),number(Y)),_,N) :- calcular(Op,X,Y,N).
 calcular_expr(expr(Op,name(X),number(Y)),Dicc,N) :- get_dict(X,Dicc,Val),calcular(Op,Val,Y,N).
 calcular_expr(expr(Op,number(X),name(Y)),Dicc,N) :- get_dict(Y,Dicc,Val),calcular(Op,X,Val,N).
 calcular_expr(expr(Op,name(X),name(Y)),Dicc,N) :- get_dict(X,Dicc,ValX),get_dict(Y,Dicc,ValY),calcular(Op,ValX,ValY,N).
 calcular_expr(expr(Op,name(X),Expr),Dicc,N) :- calcular_expr(Expr,Dicc,N1),get_dict(X,Dicc,Val),calcular(Op,Val,N1,N).
 calcular_expr(expr(Op,number(X),Expr),Dicc,N) :- calcular_expr(Expr,Dicc,N1), calcular(Op,X,N1,N).

 calcular(+,X,Y,N) :- N is X + Y.
 calcular(-,X,Y,N) :- N is X - Y.
 calcular(*,X,Y,N) :- N is X * Y.
 calcular(/,X,Y,N) :- N is X / Y.

 comparar_aux(compare(OpL,number(X),number(Y)),_) :- comparar(OpL,X,Y),!.
 comparar_aux(compare(OpL,number(X),name(Y)),Dicc) :- get_dict(Y,Dicc,Val), comparar(OpL,X,Val),!.
 comparar_aux(compare(OpL,name(X),number(Y)),Dicc) :- get_dict(X,Dicc,Val),comparar(OpL,Val,Y),!.
 comparar_aux(compare(OpL,name(X),name(Y)),Dicc) :- get_dict(X,Dicc,ValX),get_dict(Y,Dicc,ValY), comparar(OpL,ValX,ValY),!.

 comparar(=,X,Y) :- X =:= Y.
 comparar(<,X,Y) :- X < Y.
 comparar(>,X,Y) :- X > Y.
 comparar(<=,X,Y) :- X =< Y.
 comparar(>=,X,Y) :- X >= Y.

 es_cadena_char(Atomo) :- atom_chars(Atomo, [Primero|_]),Primero == '\''.
```

4.- Conclusión

Durante todo el proceso de la realización del caso de estudio, hemos estado utilizando amplios conceptos de la programación lógica al mismo tiempo que lo estábamos aplicando a otros aspectos de la computación como puede ser los compiladores, otorgándonos un mejor entendimiento de estos gracias a participar en cada fase de la compilación, desde el proceso inicial de tokenización hasta el desarrollo de la maquina abstracta y obtener una respuesta.

También hemos obtenido una mejor visión general de los lenguajes de programación imperativo, como puede ser Pacal, al necesitar saber la estructura de los programas y de sus instrucciones para poder llevar a cabo el proceso de parseamiento de los tokens y transformarlo a un árbol semántico.

Otro aspecto es que la programación lógica permite la **multidisciplinariedad** entre ramas de conocimiento, permitiéndonos como hemos comprobado aplicar conocimientos técnicos para desarrollar un concepto más completo y complejo, siendo enriquecedor para la formación del alumnado en su etapa estudiantil.

5.- Bibliografía

Sterling, L., & Shapiro, E. Y. (1994). The art of Prolog: advanced programming techniques. MIT press. "Logic Grammars" (Apartado 19.3) y Capitulo "A compiler"

Prolog: Programming For Artificial Intelligence. Third Edition (International Computer Science Series) Capitulo "Language Processing with Grammar Rules"

6.- Webgrafía

https://www.swi-prolog.org/pldoc/doc_for?object=manual