UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

COMPILADOR DE UN SUBCONJUNTO DE PASCAL

Autores:

Javier Oliver Díaz-Alejo

Mª Dolores Sesmero Pozo

ÍNDICE

A. RESUMEN DE LOS CONCEPTOS TEÓRICOS O PRÁCTICOS RELACIONADOS CON EL CASO DE ESTUDIO.

INTRODUCCIÓN

Un compilador es un pequeño programa informático, que se encarga de traducir (compilar) el código fuente de cualquier aplicación que se esté desarrollando. En pocas palabras, es un software que se encarga de traducir el programa hecho en lenguaje de programación, a un lenguaje de máquina que pueda ser comprendido por el equipo y pueda ser procesado o ejecutado por este.

Un concepto un poco más elaborado es el siguiente: Un compilador es un programa que convierte o traduce el código fuente de un programa hecho en lenguaje de alto nivel, a un lenguaje de bajo nivel (lenguaje de máquina). Los programas escritos en lenguajes de alto nivel se llaman programas fuente y El programa traducido se llama programa objeto. El compilador traduce sentencia a sentencia el programa fuente.

En el caso de que el lenguaje fuente sea un lenguaje de programación de alto nivel y el objeto sea un lenguaje de bajo nivel (ensamblador o código de máquina), a dicho traductor se le denomina compilador. Un ensamblador es un compilador cuyo lenguaje fuente es el lenguaje ensamblador. Un intérprete no genera un programa equivalente, sino que toma una sentencia del programa fuente en un lenguaje de alto nivel y la traduce al código equivalente y al mismo tiempo lo ejecuta.

Un lenguaje de programación viene definido por un léxico, una sintaxis y una semántica.

- Léxico: Conjunto de símbolos que se pueden usar en un lenguaje. Estos símbolos o elementos básicos del lenguaje, podrán ser de los siguientes:
  - Identificadores: nombres simbólicos que se darán a ciertos elementos de programación (p.e. nombres de variables, tipos, módulos, etc.).
  - Constantes: datos que no cambiarán su valor a lo largo del programa.
  - Operadores: símbolos que representarán operaciones entre variables y constantes.
  - Instrucciones: símbolos especiales que representarán estructuras de procesamiento, y de definición de elementos de programación.
  - Comentarios: texto que se usará para documentar los programas

- Sintaxis: Consta de unas definiciones, denominadas reglas sintácticas o producciones que especifican la secuencia de símbolos que forman una frase del lenguaje. Estas reglas dicen si una frase está bien escrita o no. Las reglas sintácticas pueden contener dos tipos de elementos:
  - Elementos Terminales ($\in$ Vocabulario)
  - Elementos no Terminales, que son construcciones intermedias de la gramática.

Existen diversas formas de especificar las reglas, pero únicamente nos vamos a centrar en una de ellas:

- Notación BNF (Backus-Naur Form). Es de las primeras notaciones que se empezó a utilizar para especificar lenguajes de programación.

  Notación BNF: <elemento no terminal> ::= Definición1 | Definición2 | ...

  Los elementos terminales, o sea, que pertenecen al vocabulario, se escriben tal cual. Los elementos no terminales se escriben entre los símbolos <>. Ejemplo:

Ejemplo: *Descripción sintáctica de una expresión matemática en notación BNF:*
---> *4*(3+1)*

<expresión> ::= <numero> | (<expresión>) | <expresión><operador><expresión>

<operador> ::= + | - | * | /

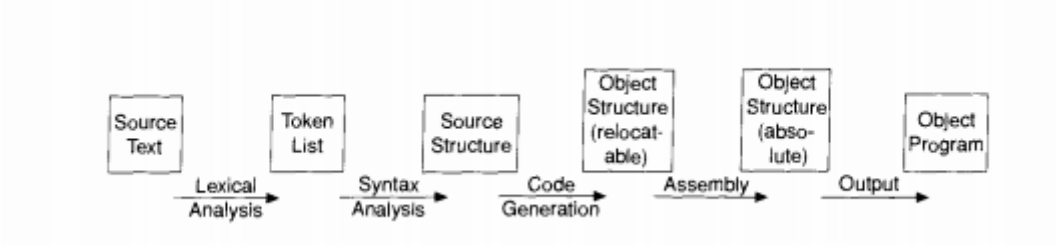
<numero> ::= <digito> | <numero><digito>

<digito> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

- Semántica: Define el significado de las construcciones sintácticas del lenguaje y de las expresiones y tipos de datos utilizadas.

ETAPAS DE COMPILACIÓN

El objetivo de un compilador es traducir un programa en lenguaje fuente a un programa en lenguaje ensamblador, para una arquitectura de computador en particular. La tarea de compilación puede dividirse en cinco etapas: análisis léxico, análisis sintáctico, generación de código, ensamblado y salida.



La primera etapa, análisis léxico, transforma el código fuente en una lista de tokens. En el análisis sintáctico se analiza la lista de tokens para generar una estructura. La tercera y cuarta etapa transforma la estructura en código reubicable y el código reubicable en código objeto absoluto respectivamente. Por último, en la quinta etapa se produce como salida el programa objeto.

Nuestro diseño se va a resumir en tres etapas: análisis léxico, análisis sintáctico y computar/ejecutar.

➢ ANÁLISIS LÉXICO

Un analizador léxico es la primera fase de un compilador consistente en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de tokens (componentes léxicos) o símbolos. Estos tokens sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico (parser).

La especificación de un lenguaje de programación a menudo incluye un conjunto de reglas que definen el léxico. Estas reglas consisten comúnmente en expresiones regulares que indican el conjunto de posibles secuencias de caracteres que definen un token o lexema.

Esta etapa está basada usualmente en una máquina de estados finitos. Esta máquina contiene la información de las posibles secuencias de caracteres que puede conformar cualquier token que sea parte del lenguaje.

➢ ANÁLISIS SINTÁCTICO

Todo lenguaje de programación tiene reglas que describen la estructura sintáctica de programas bien formados. En Pascal, por ejemplo, un programa se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o notación BNF.

Es la fase del analizador que se encarga de chequear el texto de entrada en base a una gramática dada. Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce. En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce la secuencia de tokens suministrada por el analizador léxico.

PARSER:

Un analizador sintáctico (o parser) es una de las partes de un compilador que transforma su entrada en un árbol de derivación.

El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada.

Un parser adecuado se escribe mediante una gramática de cláusulas definida (DCG)

- Una gramática cláusula definida (DCG) es una forma de expresar la gramática, ya sea para naturales o formales idiomas, en un lenguaje de programación lógica, como Prolog. Se llaman gramáticas de cláusulas definitivas, ya que representan una gramática como un conjunto de cláusulas positivas en la lógica de primer orden.
  Las cláusulas positivas de un DCG se pueden considerar un conjunto de axiomas en que la validez de una sentencia, y el hecho de que tiene un cierto árbol de análisis pueden considerarse teoremas que se derivan de estos axiomas.
  Ejemplo de gramática DCG:

```
s → a(N), b(N), c(N).

a(N) → [a], a(N1), {N is N1+1}.
a(0) → [ ].
b(N) → [b], b(N1), {N is N1+1}.
b(0) → [ ].
c(N) → [c], c(N1), {N is N1+1}.
c(0) → [ ].
```

Program 19.2 Recognizing the language $a^N b^N c^N$

La primera declaración de cualquier programa en pascal debe de ser una declaración de programa, que consiste en la palabra “programa” seguida del nombre del programa. El nombre de un programa es un identificador en el idioma. El nombre del programa es seguido por un “;”, otro identificador estándar, y seguido de la palabra “begin”. El cuerpo de un programa pascal consiste en declaraciones, o más preciso, en una simple declaración que a su vez podría consistir en varias declaraciones. Todo esto se resume en la regla gramatical de nivel superior:

```
pl_program(S) →
    [program], identifier(X), [';'], statement(S).
```

La estructura devuelta como salida del parsing es la declaración que constituye el cuerpo del programa. La primera declaración que describimos es una sentencia compuesta. Su sintaxis es el identificador estándar ”begin” seguido de la primera declaración, S, por ejemplo, en la sentencia compuesta, y luego las restantes declaraciones Ss.

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

Las declaraciones en pascal están delimitadas por un “;”. El resto de declaraciones son definidas como un “;” seguido de una declaración y recursivamente del resto de declaraciones.

```
rest_statements((S;Ss)) →
    [';'], statement(S), rest_statements(Ss).
```

El final de una secuencia de declaraciones se indica por el identificador estándar “end”.

```
rest_statements(void) → [end].
```

La anterior definición de declaración excluye la posibilidad de una declaración vacía. Hay muchos tipos de declaraciones, de los cuales nosotros vamos a tratar en nuestro programa pascal las siguientes:

```
statement(assign(X,E)) →
    identifier(X), [':='], expression(E).


  statement(if(T,S1,S2)) →
      [if], test(T), [then], statement(S1),
      [else], statement(S2).

test(compare(Op,X,Y)) →
   expression(X), comparison_op(Op), expression(Y).

statement(while(T,S)) →
    [while], test(T), [do], statement(S).

statement(read(X)) → [read], identifier(X).
statement(write(X)) → [write], expression(X).
```

- **COMPUTAR/EJECUTAR**

Una vez realizado el análisis sintáctico y obtenido una estructura el último paso consiste en ejecutar dicha estructura para obtener los resultados necesarios, es decir, someter la entrada hacia la computadora, la cual transforma en una salida de acuerdo con las instrucciones en el programa.

B. DOCUMENTACION DEL PROGRAMA

INTRODUCCION

El compilador para un subconjunto de pascal ha sido programado en prolog tal y como se nos pedía en el enunciado de su realización.

Como ya mencionamos anteriormente la construcción del compilador ha sido dividida en 3 fases, por lo que en su implementación hemos extrapolado esto al programa de forma que la compilación se hace mediante 3 predicados generales:

- **analisisLexico(+Ar,-Ltok),** que recibe una ruta a un archivo mediante un string Ar y lo trasforma en una lista de tokens Ltok.
- **parse(+Source, -Structure, -Variables),** que recibe una lista de tokens (source) y la transforma en una estructura que permita la ejecución del programa además de también sacar un estado de las variables con sus valores correspondientes, que se irá modificando en la etapa de ejecución cuando sea correspondiente.
- **ejecutar**, que tiene varias implementaciones según la necesidad del momento, es el predicado que toma la estructura y el estado de las variables y consigue simular la ejecución del programa en una pseudomaquina virtual.

Estos 3 predicados generales pueden subdividirse en otros predicados que a continuación explicaremos.

ANALISIS LEXICO

Para el análisis léxico hemos creado el predicado girando alrededor de la herramienta para tokenizar propia de prolog que es el predicado **Split_string/4**.

Esta parte se basa esencialmente en el predicado **analisisLexico/2,** este a la vez invoca a **tokenize_file(Ar, Ltok).** En **tokenize_file/2** comenzamos a tokenizar el archivo basándonos en varios predicados:

- **open/3** que coge un string de ruta a un archivo y según las opciones te saca un stream de lectura/escritura.
- **Read_string/5** de un stream te saca un string o lista de strings según los caracteres que pongas de corte y los que omitas.
- **Close/1** cierra el stream para acabar su uso
- **String_chars/2** te divide un string en una lista de caracteres.
- **Limpiartokens/2**, es un predicado que hemos creado y lo que hace es recorrer la lista de caracteres recibida y prepararla para su posterior procesamiento, detectando símbolos y aislándolos.

- **Quitarrepes/2**, otro predicado que transforma el pre tratamiento de limpiartokens a una lista tratable por el Split_string.
- **Atomic_list_concat/2**, un predicado propio de prolog que nos permite dada una lista de caracteres o de átomos transfórmala a un string completo.
- **Split_string/4,** el predicado final que nos permite tokenizar el texto del archivo, basándonos en el pre tratamiento y en los caracteres que sí y no nos interesan podemos hacer un Split al String para que nos dé justo los tokens que necesitamos.

En este apartado incluimos una restricción a la hora de programar en el subconjunto que compilamos, estar restricción dice que no puedes dejar líneas en blanco en tu código, esto se debe a que si introduces una línea de código vacía el número de caracteres \n se vuelve intratable y siempre queda un token vacío.

ANÁLISIS SINTÁCTICO

En la creación de la estructura y el análisis sintáctico hemos usado un parser modificado para nuestro propósito, ya que el parser (basado en el libro The art of prolog) contemplaba un subconjunto de pascal demasiado pequeño para tal.

Nuestro predicado parse/3 se basa en las reglas DCG. Estas reglas semánticas nos indican de que están formados una serie de elementos y basándonos en ellas hemos creado una regla general que nos indica de que se forma el programa, la cual a su vez se apoya en otras reglas que nos transforman los tokens en una estructura.

Las reglas se muestran a continuación:

- **Pl_program**, regla general, nos indica que un programa se compone de la palabra “program” más un identificador, más un “;” más un bloque de declaraciones y terminando el programa en un punto.
- **Bloque**, esta regla nos dice que un bloque se forma de una declaración de variables más una o más sentencias o instrucción.
- **Declaración**, nos dice que una declaración de variables se forma por la palabra “var” más 0 o más variables.
- **Variable**, nos dice que una variable (declarada) se forma por un identificador, más el carácter “:” más la palabra “Integer”, ya que en este caso nuestro programa trata sólo enteros.
- El **resto de variable** puede ser o un punto y coma (no hay más variables u otra declaración, es decir, un punto y coma más otra variable más otro resto de variables.
- **Statement,** esta regla nos dice que puede formar una sentencia, contemplamos las estructuras while, if, asignación, lectura y escritura, además de una general que forma la estructura que nos dice que una sentencia o grupo de sentencias se forman con la palabra “begin” seguida de

una sentencia, seguida de un conjunto de sentencias. Este conjunto de sentencias puede ser la palabra “end”, (que marca el fin del bloque) u otras sentencias.

Gracias a los argumentos en las reglas semánticas y la función de prolog que nos permite dentro de las sentencias operar con elementos, conseguimos construir la estructura y el estado de las variables.

EJECUCION

En la ejecución y su implementación hemos girado en torno a las reglas DCG creadas en la etapa de parser y análisis semántico. No hemos creado más reglas DCG, pero si haciendo un proceso de “inversión” de las mismas hemos conseguido realizar la ejecución.

Este proceso de inversión es un proceso mediante el cual ejecutamos la estructura contando con los valores de las variables en el estado, este proceso inverso se logra gracias a nuestro conocimiento a priori de la estructura y de cómo se ha formado, por lo que hemos creado unos predicados **ejecutar** acordes a las reglas DCG.

Para acompañar la ejecución hemos creado una serie de predicados que nos ayudan en la misma:

- **Escribir**, que nos escribe el nombre de una variable con un igual y el valor de dicha variable en el estado.
- **Calcular expresión**, un predicado que nos permite dada una expresión calcularla en función de si tiene en ella cualquier combinación de valor inmediato y variables.
- **Operar**, para simplificar el predicado calcular expresión hemos creado el predicado operar que nos permite dados dos valores operarlos en función de la operación que los relaciona.
- **Comprobar,** este predicado nos permite dada una comparación de variables o valores, busca dichos valores en el estado actual de dicha variable y retornar true si la comparación de dichos valores se cumple y false en caso contrario.
- **Buscar valor**, es un predicado que dado el nombre de una variable nos retorna su valor dentro del estado.
- **Asignar Valor**, es un predicado que, dado un valor y un nombre de variable, asignara en el estado de las variables tal valor a tales variables.

Tras la implementación de la ejecución nos dimos cuenta de las limitaciones de nuestro sistema, en este caso añadimos una nueva limitación que nos dice que una expresión o una operación de asignación, solo puede tener dos valores, es decir, solo puede multiplicar/dividir/sumar/restar dos valores el uno con el otro.

Tras la implementación de esta parte nuestro compilador está finalizado con una pseudomaquina virtual que simula la ejecución del código compilado.

RESTRICCIONES

A lo largo de la implementación hemos ido asignando restricciones al subconjunto de Pascal y a la escritura del código que podemos "compilar", algunas han sido mencionadas otras no, aquí dejamos un listado de todas las restricciones:

- No se pueden crear un código en pascal con líneas en blanco
- No se pueden ejecutar operaciones de más de dos operando
- No se pueden usar variables que no hayan sido declaradas
- Cuando se escribe un valor por pantalla se debe escribir el valor seguido de un punto, es decir, si se introducen (como permite la interfaz de prolog) varios números a la vez la ejecución fallara.

C. CÓDIGO FUENTE DEL PROGRAMA

```
tokenize_file(File, Tokens) :-
   open(File, read, F),read_string(F, "","",S, String), close(F), string_chars(String,T),
        limpiartokens(T,SL), quitarrepes(SL,SL2),atomic_list_concat(SL2,StringL),
split_string(StringL, "\n\n", "\t", Tokens).


limpiartokens([],[]).
limpiartokens([/|Rest],['\n',/,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([+|Rest],['\n',+,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([-|Rest],['\n',-,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([*|Rest],['\n',*,'\n'|Zy]):-limpiartokens(Rest,Zy).


limpiartokens([>,=|Rest],['\n','>=','\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([<,=|Rest],['\n','<=','\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([<|Rest],['\n',<,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([>|Rest],['\n',>,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([=|Rest],['\n',=,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([!,=|Rest],['\n','!=','\n'|Zy]):-limpiartokens(Rest,Zy).


limpiartokens([;|Rest],['\n',;,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([:,=|Rest],['\n',:=,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([:|Rest],['\n',:,'\n'|Zy]):-limpiartokens(Rest,Zy).
limpiartokens([.|Rest],['\n',.|Zy]):-limpiartokens(Rest,Zy).

limpiartokens([' '|Rest],['\n'|Zy]):-limpiartokens(Rest,Zy).


limpiartokens([X|Rest],[X|Zy]):- limpiartokens(Rest,Zy),!.


quitarrepes([],[]).
quitarrepes(['\n','\n'|Rest],['\n'|Zy]):-quitarrepes(Rest,Zy).
quitarrepes([X|Rest],[X|Zy]):-quitarrepes(Rest,Zy),!.


%PARSER



pl_program(D,E) --> ["program"], identifier(X), [";"], bloque(D,E), ["."].

bloque(D,E) --> declaracion(D), statement(E).

declaracion([V,X|R]) --> ["var"], variable(V,X), resto_variables(R).

resto_variables([V,X|R]) -->  [";"], variable(V,X), resto_variables(R).
resto_variables([]) --> [";"].
```

```
variable(V,X) --> identifier(V), [":"], ["Integer"], {X is 0}.

statement((S;Ss)) --> ["begin"], statement(S), rest_statement(Ss).
statement(read(X)) --> ["read"], identifier(X), [";"].
statement(assign(X,V)) --> identifier(X), [":="], expression(V), [";"].
statement(if(T,S1,S2)) --> ["if"], test(T), ["then"], statement(S1), ["else"],
statement(S2).
statement(while(T,S)) --> ["while"], test(T), ["do"], statement(S).

statement(write(X)) --> ["write"], expression(X), [";"].

rest_statement(end) --> ["end"].
rest_statement((S;Ss)) --> statement(S), rest_statement(Ss).

expression(X) --> pl_constant(X).
expression(expr(Op, X, Y)) --> pl_constant(X), aritmetic_op(Op), expression(Y).

aritmetic_op('+') --> ["+"].
aritmetic_op('*') --> ["*"].
aritmetic_op('-') --> ["-"].
aritmetic_op('/') --> ["/"].

pl_constant(X) --> identifier(X).

identifier(X) --> [X].

test(compare(Op,X,Y)) --> expression(X), comparison_op(Op), expression(Y).

comparison_op('=') --> ["="].
comparison_op('!=') --> ["!="].
comparison_op('>') --> [">"].
comparison_op('<') --> ["<"].
comparison_op('>=') --> [">="].
comparison_op('<=') --> ["<="].


%EJECUCION



ejecutar((S;Ss), V, NNV) :- ejecutar(S, V, NV), ejecutar(Ss, NV, NNV).
ejecutar(read(X),V, NV) :-  read(F), asignarvalor(X,F,V, NV).
ejecutar(assign(X,Y), V, NV) :- calcularExpresion(Y, V, S), asignarvalor(X, S, V, NV).
ejecutar(if(T,S1,S2),V, NV) :- comprobar(T, V),!, ejecutar(S1, V, NV).
ejecutar(if(T,S1,S2),V, NV) :- ejecutar(S2, V, NV).
ejecutar(while(T,S),V, NNV) :- comprobar(T, V),!, ejecutar(S, V, NV),
ejecutar(while(T,S),NV,NNV).
ejecutar(while(T,S),NV,NV).
ejecutar(write(X),V, V) :- escribir(X, V).
ejecutar(end, V, V).
```

```
escribir(X,V) :- buscarValor(X, V, R1), write(X), write("="), write(R1), write("\n").
escribir(X,V) :- atom_number(X, R1), write_ln(R1).

calcularExpresion(expr(X, R, Z), V, S) :- atom_number(R, R1), atom_number(Z, Z1),
operar(X, R1, Z1, S).
calcularExpresion(expr(X, R, Z), V, S) :- atom_number(R, R1), buscarValor(Z, V, Z1),
operar(X, R1, Z1, S).
calcularExpresion(expr(X, R, Z), V, S) :- buscarValor(R, V, R1), atom_number(Z, Z1),
operar(X, R1, Z1, S).
calcularExpresion(expr(X, R, Z), V, S) :- buscarValor(R, V, R1), buscarValor(Z, V,
Z1), operar(X, R1, Z1, S).
calcularExpresion(Y, V, S) :- atom_number(Y, S).

operar(X, R1, Z1, S) :- X='+', S is R1+Z1.
operar(X, R1, Z1, S) :- X='*', S is R1*Z1.
operar(X, R1, Z1, S) :- X='-', S is R1-Z1.
operar(X, R1, Z1, S) :- X='/', S is R1/Z1.


comprobar(compare(X,Y,Z), V) :- X = '=', buscarValor(Y, V, Yi), buscarValor(Z, V,
Zi), Yi=Zi.
comprobar(compare(X,Y,Z), V) :- X = '>', buscarValor(Y, V, Yi), buscarValor(Z, V,
Zi), Yi>Zi.
comprobar(compare(X,Y,Z), V) :- X = '<', buscarValor(Y, V, Yi), buscarValor(Z, V,
Zi), Yi<Zi.
comprobar(compare(X,Y,Z), V) :- X = '<=', buscarValor(Y, V, Yi), buscarValor(Z, V,
Zi), Yi=<Zi.
comprobar(compare(X,Y,Z), V) :- X = '>=', buscarValor(Y, V, Yi), buscarValor(Z, V,
Zi), Yi>=Zi.

buscarValor(Y, [X,S|R], Z) :- Y=X, Z=S.
buscarValor(Y, [X,S|R], Z) :- Y\=X, buscarValor(Y, R, Z).
buscarValor(Y, [], Z) :- fail.


asignarvalor(S, Val, [X,Y|R], V):- S==X , V=[S,Val|R].
asignarvalor(S, Val, [X,Y|R], V):- S\=X, asignarvalor(S, Val, R, E), V=[X,Y|E].
asignarvalor(S, Val, [], V):- fail.



parse(Source, Structure, Variables) :- pl_program(Structure, Variables, Source,[]), !.
analisiLexico(Ar,Ltok):- tokenize_file(Ar, Ltok).
compilar(Ar):- analisiLexico(Ar,L),parse(L,V, S), ejecutar(S, V, Fin), write("Estado: "),
write_ln(Fin), !.
```

EJEMPLO DE PROGRAMA PASCAL PARA SU PUESTA EN FUNCIONAMIENTO

```
program factorial;
    var
        value: Integer;
        count: Integer;
        result: Integer;
    begin
        read value;
        count := 1;
        result := 15;
        while count <= value do
            begin
                count := count+1;
                result := result+count;
            end
        if count != value then
            begin
                count := count+5;
                result := result+20;
                count := result+20;
            end
        else
            begin
                count := count-5;
                value := result+20;
            end
        write result;
        write count;
        write 2;
    end.
```

BIBLIOGRAFÍA

- The Art of Prolog: Advanced Programming Techniques (Logic Programming)