COMPILADOR PARA UN LENGUAJE DE PROGRAMACIÓN PASCAL REDUCIDO

Laboratorio de Compiladores e Intérpretes

10/11/2018

Bonet Peinado, Daiana — FAI-238

de la Fuente, Juan — FAI-524

{daiana.bonet,juan.delafuente}@fi.uncoma.edu.ar

Grupo 06

Índice

Capítulo 1

Gramática Generadora

En el siguiente capítulo se especifica el diseño y desarrollo de una gramática para la generación de cadenas *aceptadas* por un subconjunto acotado del lenguaje de programación *Pascal* a través de todo el proceso de análisis, desarrollo analítico de la misma que incluye las técnicas utilizadas para su abordaje, las dificultades encontradas y las limitaciones que presenta su diseño, finalizando con una gramática libre de contexto (o tipo II) escrita en **Extended Backus Naur Form** (*EBNF*) en su estilo derivado de expresiones regulares.

Motivación

La realización de esta gramática es el primer paso para comenzar el diseño, y en trabajos futuros, realizar la implementación de un compilador del subconjunto de lenguaje especificado. Para realizar este diseño presentamos una gramática cuya producción sean todas las cadenas bien formadas del lenguaje, dando así el puntapié inicial al diseño del compilador.

En el diseño del compilador, se establece como forma de trabajo, una *Traducción Dirigida por Sintaxis* (*TDS*).

La *TDS* es una técnica que, en base a la especificación sintáctica brindada por una *Gramática Libre de Contexto* (que puede ser especificada en *Backus Naur Form* como veremos más adelante) ayuda a guiar la traducción de los programas al añadir reglas semánticas o fragmentos de código, a las reglas de producción de la misma.

Motivados por esta técnica, se realiza la especificación detallada a continuación, buscando lograr una gramática reducida en ambigüedad al ser específica en precedencia y asociación de operadores.

Asimismo, este trabajo forma la primer parte del desarrollo teórico de la primera práctica profesional supervisada de la carrera *Licenciatura en Ciencias de la Computación* en la formación de grado de los autores del mismo.

Se espera que el resultado, sea el comienzo efectivo en el diseño de un compilador propio del subconjunto *Pascal*, al alimentar el analizador léxico con todas las secuencias de cadenas bien formadas del Lenguaje, permitiendo en una etapa posterior, la creación de árboles de análisis sintáctico.

DESCRIPCIÓN DEL PROBLEMA

Como fue mencionado con anterioridad, el objetivo particular de este primer trabajo del diseño del compilador, radica en la definición de una gramática generadora de sentencias válidas del subconjunto del lenguaje Pascal.

La especificación de este subconjunto está basada en las siguientes reglas (conservando la sintaxis de Pascal:

- definición de variables y subprogramas.
- tipos de datos simples, únicamente: enteros y booleanos.
- constantes definidas por el lenguaje: true,false.
- subprogramas: procedimientos y funciones, solamente con pasaje de parámetros por valor.
- sentencias: de asignación, alternativa *(if then else)*, repetitiva *(while)* y sentencia compuesta *(begin end)*.
- procedimientos de entrada/salida para un solo valor *(read y write)*.
- expresiones aritméticas con los operadores +,- (unario y binario), *, / y los paréntesis (,).
- expresiones booleanas con los operadores **AND**, **OR** y **NOT** y los operadores de comparación >, <, =, ≤, ≥ y ≠ entre expresiones aritméticas. Operadores de comparación entre = y ≠ expresiones booleanas.

ESPECIFICACIÓN DEL METALENGUAJE

Para la gramática especificada se tienen en cuenta las siguientes reglas de notación que se añaden a *EBNF*:

- Los símbolos no terminales se especifican entre los símbolos '<' y '>'. Por ejemplo: *<sentencia>*
- Los símbolos terminales se especifican entre comillas simples ('). Por ejemplo: *'*'*
- En las reglas de generación α y β se encuentran separados por los símbolos '::=' que sustituyen a $\rightarrow$

- Para mayor claridad, los símbolos de los operadores de *clausura*, *estrella de Kleene* (**+** y ***) y sus auxiliares, tales cómo los paréntesis '**(**' ')' se escriben en **negrita**. Asimismo, abusando de la notación, para denotar la opcionalidad de un símbolo no terminal se utiliza **?** también con el mismo estilo.

También es necesario remarcar que el estilo elegido para representar en *EBNF*, es el *derivado de expresiones regulares* al ser el más similar en forma y sintaxis al propio de las *Gramáticas* vistas durante el transcurso de la carrera.

ESTRATEGIAS UTILIZADAS PARA LA RESOLUCIÓN DEL PROBLEMA

Para comprender el camino recorrido hasta llegar a la gramática lograda, es necesario analizar brevemente la capacidad expresiva del lenguaje que debemos generar. Para esto existen dos enfoques ampliamente utilizados: *top-down* y *bottom-up*.

En este contexto, un análisis *top-down* o *descendente* sobre las estructuras que presenta la sintaxis *Pascal*, puede resultar mas natural, ya que es sintácticamente similar a las estructuras del lenguaje de programación. Todo programa *Pascal* está comprendido entre el encabezado 'program' acompañado del nombre del programa, la definición de variables y subprogramas y todo lo que se ejecuta en el mismo rodeado de un 'begin' y 'end.' (lo que podría llamarse una expresión compuesta con la particularidad del punto final).

Esta estructura básica representa de forma genérica también la definición de los subprogramas, ya que únicamente cambia el encabezado por uno particular para cada tipo (*procedure* o *function*) que incorpora además la definición de parámetros formales.

Estructuras más sencillas, como las alternativas o las repetitivas condicionales, nuevamente tienen un encabezado particular (*if-then* o *if-then-else* para las alternativas y *while-do* para las repetitivas) sin embargo en su interior el bloque de sentencias se encuentra delimitado nuevamente por las palabras claves 'begin' y 'end'.

Cada una de las estructuras mencionadas anteriormente, contienen en los bloques de sentencias, posibles llamadas a subprogramas o sentencias sencillas del lenguaje.

Fácilmente se puede lograr una abstracción para este subconjunto limitado de *Pascal*, si se considera que cada encabezado puede ser representado por un símbolo único particular, el resto de la generación es por bloques de sentencias anidados y delimitados por las palabras reservadas 'begin'-'end'. Esto permite observar que no existe una *correspondencia cruzada* característica de los *Lenguajes Sensibles al Contexto* y delimita considerablemente el universo de Lenguajes.

Por otra parte, la necesidad de anidar bloques 'desde afuera hacia dentro' demarca la posibilidad de que exista una *correspondencia por capas* propia de los *Lenguajes Libres de Contexto*. Cabe aclarar que esta correspondencia responde a la forma $a^n c b^n, n \geq 0$. [5]

Para demostrar que la gramática maximal para el lenguaje *Pascal*, es *Libre de Contexto*, basta con recurrir al *Teorema de Pumping para Lenguajes Regulares* que enuncia: [3]

'Sea L un lenguaje regular infinito. Entonces existe un entero $n \geq 1$ tal que cualquier cadena $w \in L$ con $|w| \geq n$ puede ser escrita como $w = xyz$ tal que $y \neq \lambda$, $|xy| \leq n$ y para todo $k \geq 0, xy^k z \in L$' [6]

Para el caso más sencillo aceptado por el lenguaje, consideremos el bloque vacío de sentencias:

```
PROGRAM programa;
BEGIN

END.
```

Luego, podemos definir recursivamente que dentro de ese bloque, puede haber n bloques sin sentencias anidados (considerando que se recurren a estructuras sencillas que para mayor claridad se omite su definición), que siguen siendo sentencias válidas de la sintaxis de *Pascal*.

```
PROGRAM programa;
BEGIN
    BEGIN
            ...
    END;
END.
```

Por cuestiones de claridad en la prueba, vamos a considerar dividir este programa básico en dos lenguajes: L_1 =PROGRAM y L_2 =BEGIN^n END^n con $n \geq 0$ a sabiendas que los lenguajes regulares son cerrados bajo unión (de forma similar se podría incorporar el símbolo '.' faltante al concatenarse cómo lenguaje regular ya que los lenguajes finitos son regulares).

Luego, L_2 =BEGIN^n END;^{n} con $n \geq 0$ puede re-escribirse (considerando la transformación como un homomorfismo simple) como L_2 =x^n y^n con $n \geq 0$. Por último, analizamos *Pumping Theorem* para *Lenguajes Regulares* sobre L_2:

Teorema 1 *Si la cadena $w = x^n y^n \in L$, con $|w| \geq n$, entonces por Teorema, puede ser escrita como $w = xyz$ tal que $|xy| \leq n$ e $y \neq \lambda$, esto es $y = a^i$ para algún $i > 0$.*

Pero entonces $xy^0 z = xz = a^{n-i} \notin L$, lo que contradice el Teorema, y por lo tanto L no es Regular.

Sabiendo que tratamos con un *Lenguaje Libre de Contexto*, podemos recurrir al metalenguaje *Backus Naur Form* en su versión extendida sin temor a perder expresividad de lenguaje, ya que son equivalentes en poder denotacional a las *Gramáticas Libres de Contexto*. [5]

Luego, para lograr las reglas de producción correctas, fue necesario analizar en profundidad las posibles combinaciones de operandos entre expresiones y su precedencia, lo que derivó en una generación ordenada de expresiones que parten desde el término *Expresiones Generales*.

En simultáneo, fue posible definir la estructura de programas y subprogramas compuestos por los bloques de sentencia antes mencionado, así como se replicó su diseño (específico para la variación necesaria) en las estructuras de control y repetición.

LIMITACIONES

Las limitaciones que encontramos a la hora de realizar este trabajo pueden ser clasificadas en dos tipos:

- Generadas por la gramática desarrollada
- Generadas por el subconjunto del lenguaje utilizado

Generadas por la gramática

Estas limitaciones son consecuencia de las decisiones de diseño que debieron tomarse, para abordar la resolución del problema planteado.

En particular, se limitó cierta flexibilidad que presentaba la sintaxis sobre las palabras claves en grafía minúscula. Si bien es cierto que el lenguaje en su especificación hace distinción de mayúsculas, la gramática fue definida de manera tal que toda *palabra clave* representada en símbolos terminales se encuentra únicamente en un tipo de grafía. Esto fue considerado para lograr mayor legibilidad al momento de analizar la construcción de la gramática, sin embargo, sería necesario añadir las reglas de generación, o realizar una abstracción acorde para que pueda reconocer sin distinción de mayúsculas y minúsculas.

Quedará como trabajo a futuro, lograr una mayor flexibilidad en la generación de las *palabras claves* y así contar con una gramática más apegada a la especificación formal de *Pascal*.

Generadas por el subconjunto del lenguaje

Al tomar un subconjunto del lenguaje Pascal, se logra reducir el tamaño del desafío planteado para la definición de la gramática teniendo como consecuencia la notable reducción en la versatilidad y potencia del lenguaje. En este sentido podemos demarcar las siguientes limitaciones, entre otras:

- Tipos de Datos definidos por el usuario.
- Constantes.
- Arreglos y Estructuras n-dimensionales.
- Punteros.
- Definición de Rangos y Conjuntos por extensión.
- Estructuras Repetitivas Incondicionales.

Estos son sólo algunos ejemplos específicos de la pérdida de poder expresivo de la definición del lenguaje *Pascal*. Sin embargo, trabajar con un subconjunto limitado del mismo, permite lograr una mayor comprensión del proceso de diseño de un compilador, cuyo verdadero objetivo es el destino de estas prácticas.

DEFINICIÓN DE LA GRAMÁTICA

Sea la gramática G definida como:

$$G = \{V_n, V_t, S, P\}$$

donde

V_n representa el conjunto finito de símbolos no terminales o auxiliares
V_t es el conjunto finito de símbolos terminales
S es el símbolo inicial de la gramática
P es un conjunto finito de reglas de producción de la forma $\alpha \rightarrow \beta$
Para este caso particular se define que,
V_n = { <*llamadaProcedimiento*>,<*parametrosReales*>,<*parametrosFormales*>,<*declaracionPyf*>,
<*while*>,<*else*>,<*ifthen*>,<*expresion*>,<*compararAnd*>,<*expresionGeneral*>,<*programa*>,
<*operadorRelacional*>,<*factor*>,<*termino*>,<*expresionAritmetica*>,<*asignacion*>,<*sentencia*>,

<fin>,<sentenciaCompuesta>,<tipoVariables>,<listaIdentificador>,<listaVariables>,
<declaracionVariables>,<identificador>,<digitos>,<digito> }

V_t = { *'a'* , *'b'* , *'c'* , ... , *'A'* , *'B'* , *'C'* , ... , *'Z'* , *'0'* , *'1'* , *'2'* , ... , *'9'* , *'and'* , *'begin'* , *'do'* , *'else'* , *'end'* , *'false'* , *'function'* , *'if'* , *'integer'* , *'procedure'* , *'program'* , *'read'* , *'then'* , *'true'* , *'var'* , *'while'* , *'write'* ,'+','−','∗','\',' <',' >',' <>',' =',' <=',' >=',' :=',' :','(',')',',',';','.' }

S = *<programa>*

P es un conjunto de reglas detalladas a continuación

Reglas de Producción P

⟨*programa*⟩ ::= 'Program' ⟨*identificador*⟩ ' ; ' ⟨*declaracionVariables*⟩**?** **(**⟨*declaracionPyf*⟩**)*** 'begin' ⟨*sentenciaCompuesta*⟩* 'end.'

⟨*identificador*⟩ ::= ⟨*letra*⟩⁺ **(**⟨*digito*⟩* ⟨*letra*⟩***)***

⟨*declaracionVariables*⟩ ::= 'var' **(**⟨*listaVariables*⟩' ; ' **)**⁺

⟨*listaVariables*⟩ ::= ⟨*listaIdentificador*⟩ ' : ' ⟨*tipoVariable*⟩

⟨*listaIdentificador*⟩ ::= ⟨*identificador*⟩ **(**, ⟨*identificador*⟩**)***

⟨*tipoVariables*⟩ ::= 'integer'
| 'boolean'

⟨*declaracionPyf*⟩ ::= 'procedure' ⟨*identificador*⟩ ⟨*parametrosFormales*⟩*' ; ' ⟨*declaracionVariables*⟩**?** **(**⟨*declaracionPyf*⟩**)*** ⟨*sentenciaCompuesta*⟩
| 'function' ⟨*identificador*⟩ ⟨*parametrosFormales*⟩*' : ' ⟨*tipoVariables*⟩ ' ; ' ⟨*declaracionVariables*⟩**?** **(**⟨*declaracionPyf*⟩**)*** ⟨*sentenciaCompuesta*⟩

⟨*parametrosFormales*⟩ ::= '('⟨*listaIdentificador*⟩ ' : ' ⟨*tipoVariable*⟩ **(**' ; '⟨*parametrosFormales*⟩**)***')'

⟨*sentenciaCompuesta*⟩ ::= 'begin' **(** ⟨*compuesta*⟩ **)*** 'end' ' ; '

⟨*compuesta*⟩ ::= ⟨*sentencia*⟩ **(** ' ; ' ⟨*compuesta*⟩**)***
| **(** ⟨*sentenciaCompuesta*⟩**)***

⟨*sentencia*⟩ ::= ⟨*ifthen*⟩
| ⟨*while*⟩
| ⟨*identificador*⟩ ⟨*asignacion*⟩
| ⟨*identificador*⟩**?** ⟨*llamadaProcedimiento*⟩

⟨*llamadaProcedimiento*⟩ ::= ‘(’⟨*expresionGeneral*⟩⟨*parametrosReales*⟩‘)’**?**
| ‘`write(`’⟨*expresionGeneral*⟩ ‘)’
| ‘`read(`’ ⟨*identificador*⟩ ‘)’

⟨*parametrosReales*⟩ ::= **(**‘,’⟨*expresionGeneral*⟩**)***

⟨*asignacion*⟩ ::= ‘`:=`’ ⟨*expresionGeneral*⟩

⟨*expresionGeneral*⟩ ::= ⟨*expresionGeneral*⟩ ‘`or`’ ⟨*compararAnd*⟩
| ⟨*compararAnd*⟩

⟨*compararAnd*⟩ ::= ⟨*compararAnd*⟩ ‘`and`’ ⟨*expresion*⟩
| ⟨*expresion*⟩

⟨*expresion*⟩ ::= ‘`not`’**?**⟨*expresionAritmetica*⟩**(**⟨*operadorRelacional*⟩⟨*expresionAritmetica*⟩**)?**

⟨*ifthen*⟩ ::= ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ⟨*compuesta*⟩ ‘`end`’ ‘`;`’
| ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ⟨*compuesta*⟩ ‘`end`’‘`else`’ ⟨*compuesta*⟩
| ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ‘`begin`’ ⟨*compuesta*⟩ ‘`end`’ ‘`;`’
| ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ‘`begin`’ ⟨*compuesta*⟩ ‘`end`’ ‘`else`’ ⟨*compuesta*⟩

⟨*mientras*⟩ ::= ‘`while`’ ⟨*expresionGeneral*⟩ ‘`do`’ ‘`begin`’ ⟨*compuesta*⟩ ‘`end`’
| ‘`while`’ ⟨*expresionGeneral*⟩ ‘`do`’ ⟨*sentencia*⟩

⟨*termino*⟩ ::= ⟨*termino*⟩ ‘`*`’ ⟨*factor*⟩
| ⟨*termino*⟩ ‘`/`’ ⟨*factor*⟩
| ⟨*factor*⟩

⟨*factor*⟩ ::= ⟨*identificador*⟩
| ⟨*llamadaProcedimiento*⟩

| ⟨*digitos*⟩
| 'true'
| 'false'
| '('⟨*expresionGeneral*⟩')'

⟨*expresionAritmetica*⟩ ::= ⟨*expresionAritmetica*⟩ '+' ⟨*termino*⟩
| ⟨*expresionAritmetica*⟩ '-' ⟨*termino*⟩
| ⟨*termino*⟩

⟨*digitos*⟩ ::= '+' ⟨*digito*⟩$^+$
| '-' ⟨*digito*⟩$^+$

⟨*letra*⟩ ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

⟨*digito*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

⟨*operadorRelacional*⟩ ::= '=' | '<>' | '<' | '<=' | '>' | '>='

PROBLEMAS ENCONTRADOS

A la hora de analizar y desarrollar la gramática antes mencionada, nos hemos encontramos con una serie de inconvenientes.

La mayor dificultad hallada fue el escaso conocimiento sobre el lenguaje *Pascal* por parte de los autores, lo que hizo necesario un estudio previo sobre la sintaxis del lenguaje.

En particular la definición de '*expresiones generales*', que es utilizada para construir sentencias lógicas empleadas en el lenguaje. Necesitó ser minuciosamente definida para el subconjunto planteado debido a la existencia de una combinación en la precedencias de operadores donde las *expresiones generales* se encuentran relacionadas con los operadores lógicos (**or**, **and**, **not**) y operadores de comparación (**'='**, **'<>'**), mientras que las *expresiones aritméticas* se encuentran relacionas a través de los operadores de comparación (**'>'**, **'<'**, **'='**, **'≤'**, **'≥'** y **'<>'**) . Este inconveniente no se encuentra en la expresividad normal de *Pascal* ya que es posible aplicar algunos operadores lógicos sobre expresiones aritméticas de forma binaria

(expresividad perdida en el subconjunto), esta virtud permite tener una única definición de *expresión aritméticas* sin necesidad de general luego las *expresiones lógicas* .

CONCLUSIONES

Analizando el trabajo realizado en su contexto, es posible afirmar que la gramática generada y debidamente justificada, marca el comienzo en el proceso de diseño de un compilador específico para *Pascal reducido*. Posicionándonos en la primer fase de la etapa de Análisis -realizado sobre el programa fuente- al permitir producir todas las cadenas legales del Lenguaje.

El resultado de ésta especificación, será el punto de inicio para la realización del próximo trabajo: **análisis léxico** mediante la construcción *árboles de análisis sintáctico*.

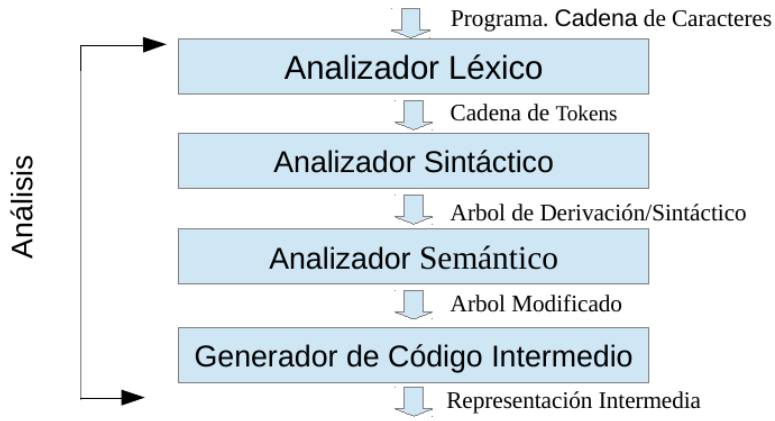

Fig. 1.1: Proceso de Análisis en el diseño de Compiladores

Como se puede observar en la figura, la entrada del *Analizador Léxico* es en sí misma, las cadenas de caracteres del programa, en otras palabras, las cadenas de caracteres generadas por la gramática realizada.

Por lo que el resultado logrado en este trabajo, es el comienzo de diversos trabajos posteriores.

Capítulo 2

Especificación Léxica

En el siguiente capítulo se detalla el diseño del **analizador léxico** encargado de identificar *tokens*, que serán suministrados al *analizador sintáctico*.

Además el analizador *léxico* en esta etapa es capaz de detectar *comentarios, tabulaciones, espacios blancos, caracteres de control.*

Por este motivo en este capítulo se definirán los posibles tokens, el patrón que representa. Tanto en una tabla de tokens como en una maquina de estados.

DESCRIPCIÓN DEL PROBLEMA

Considere el subconjunto de Pascal con las siguientes características:

- comentarios encerrados entre los caracteres '{' y '}'
- las características enunciadas en la sección *Descripción del Problema* del Capítulo 1

(a) Identifique el alfabeto de entrada.
(b) Identifique lexemas y componentes léxicos (*tokens*).
(c) Describa los patrones como expresiones regulares.
(d) Diseñe un autómata finito (o un diagrama de transición de estados) que reconozca los *tokens* correspondientes.

ESTRATEGIAS UTILIZADAS PARA LA RESOLUCIÓN DEL PROBLEMA

Alfabeto de Entrada

Se utilizará de la definición de la gramática provista en el capítulo 1, los siguientes símbolos no terminales para facilitar la definición del alfabeto Σ para la especificación léxica.

$$\textbf{LETRA} = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z\}$$

$$\textbf{DIGITO} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\textbf{SIMBOLOS} = \{-, +, *, /, =, \backslash n, \backslash t, ., (,), ;, :, \{, \}, <, >, \_\}$$

$$\Sigma = \{\textbf{LETRA} \cup \textbf{DIGITO} \cup \textbf{SIMBOLOS}\}$$

Lexemas y Componentes léxicos

Todos los *Tokens* están representados en una tabla junto al *Patrón* que los representa y algún lexema de ejemplo. El token *ESPACIO_BLANCO* es distinto a los demás porque cuando se lo reconoce, no se lo regresa al analizador sintáctico, como si ocurre con los demás *tokens*, sino que se reinicia el analizador léxico a partir del siguiente carácter luego del espacio en blanco.

Para el reconocimiento de los *Tokens* se diseñó una máquina de estados que nace de la traducción de la *expresión regular* o *patrón* a un diagrama de flujo estilizado.

Para comprender éste diagrama en profundidad, es necesario tener en cuenta las siguientes consideraciones:

- El estado q_0 representa el estado inicial de la máquina
- Cada * al lado de un estado final significa que el *"apuntador avance"* debe retroceder una posición -el lexema no incluye el ultimo símbolo consumido-
- Para representar una acción posible luego de aceptar un lexema, se detalla del lado derecho del estado final de la siguiente manera:*return(Token,valorDeAtributo)*.

Finalmente, por cuestiones de legibilidad se dividió la máquina en más de una grafo manteniendo el estado inicial en q_0 con el fin de generar una continuidad.

Lexema de ejemplo	Patron	Token
,	,	COMA
;	;	PUNTO_COMA
:	:	DOS_PUNTOS
:=	:=	ASIGNACION
.	.	PUNTO
(	(	PARENTESIS_A
)	)	PARENTESIS_C
[	[	CORCHETE_A
]	]	CORCHETE_C
-	- | +	OPERADOR_ARITMETICO
*	* | /	OPERADOR_TERMINO
<	< | <= | <> | = | > | >=	OPERADOR_RELACIONAL
and	and	AND
or	or	OR
not	not	NOT
begin	begin	BEGIN
‘prueba’	(a..z){{“_”}* {a..z | 0..9}$^{+}$}*	IDENTIFICADOR
‘10’	(0..9)$^{+}$	NUMERO
do	do	DO
else	else	ELSE
end	end	END
false	false	FALSE
function	function	FUNCTION
if	if	IF
integer	integer	INTEGER
boolean	boolean	BOOLEAN
procedure	procedure	PROCEDURE
program	program	PROGRAM
read	read	READ
then	then	THEN
true	true	TRUE
var	var	VAR
while	while	WHILE
write	write	WRITE

Table 2.1: Tabla de Tokens

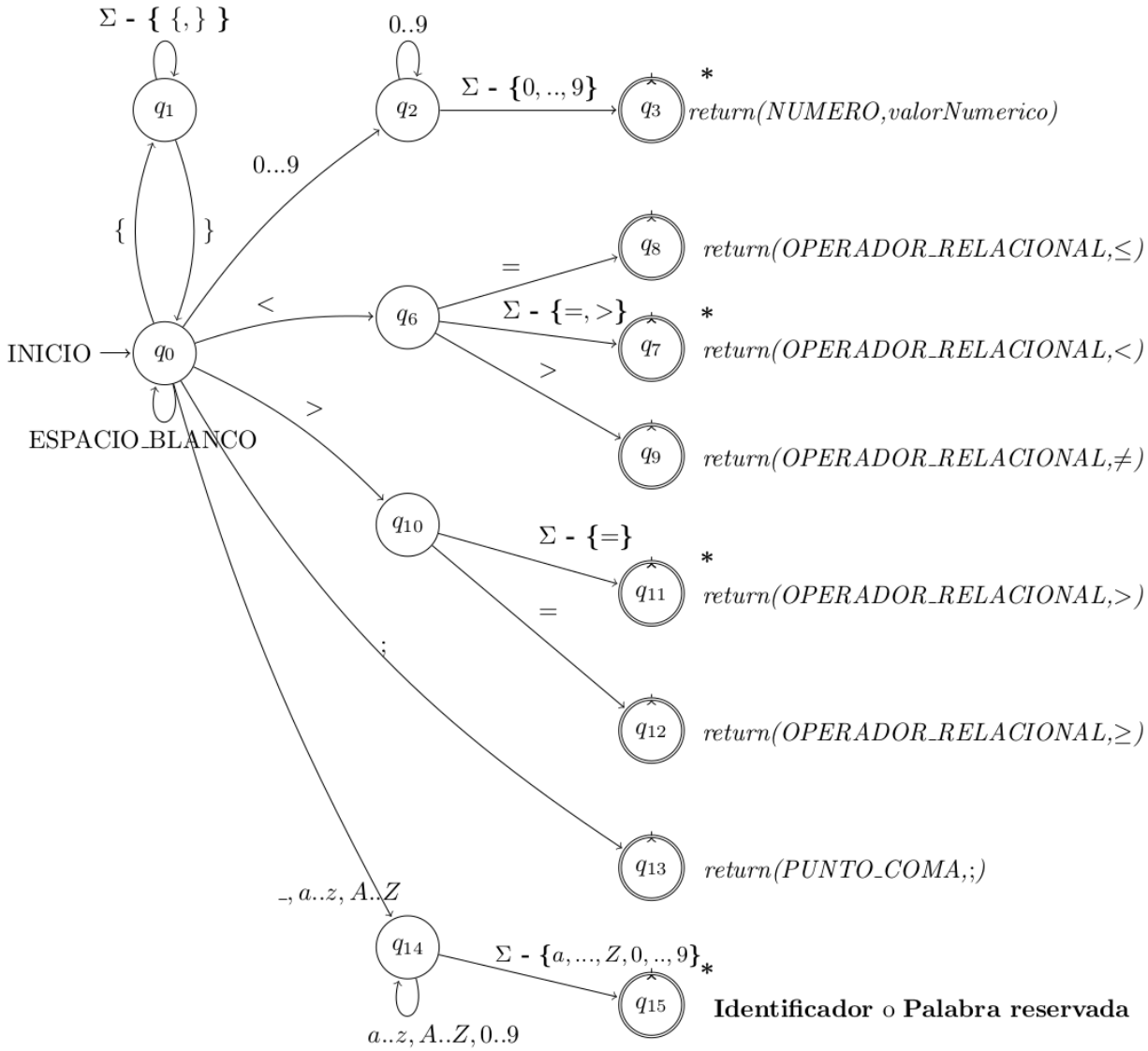

Fig. 2.1: Máquina de Estados para el reconocimiento de tokens

Identificador representa el retorno del *Token IDENTIFICADOR* descripto en la tabla correspondiente -*return(IDENTIFICADOR,valorIdentificador)*-, mientras que *Palabra reservada* simula el fragmento de la *Máquina de Estados* que vemos al pie de la siguiente hoja.

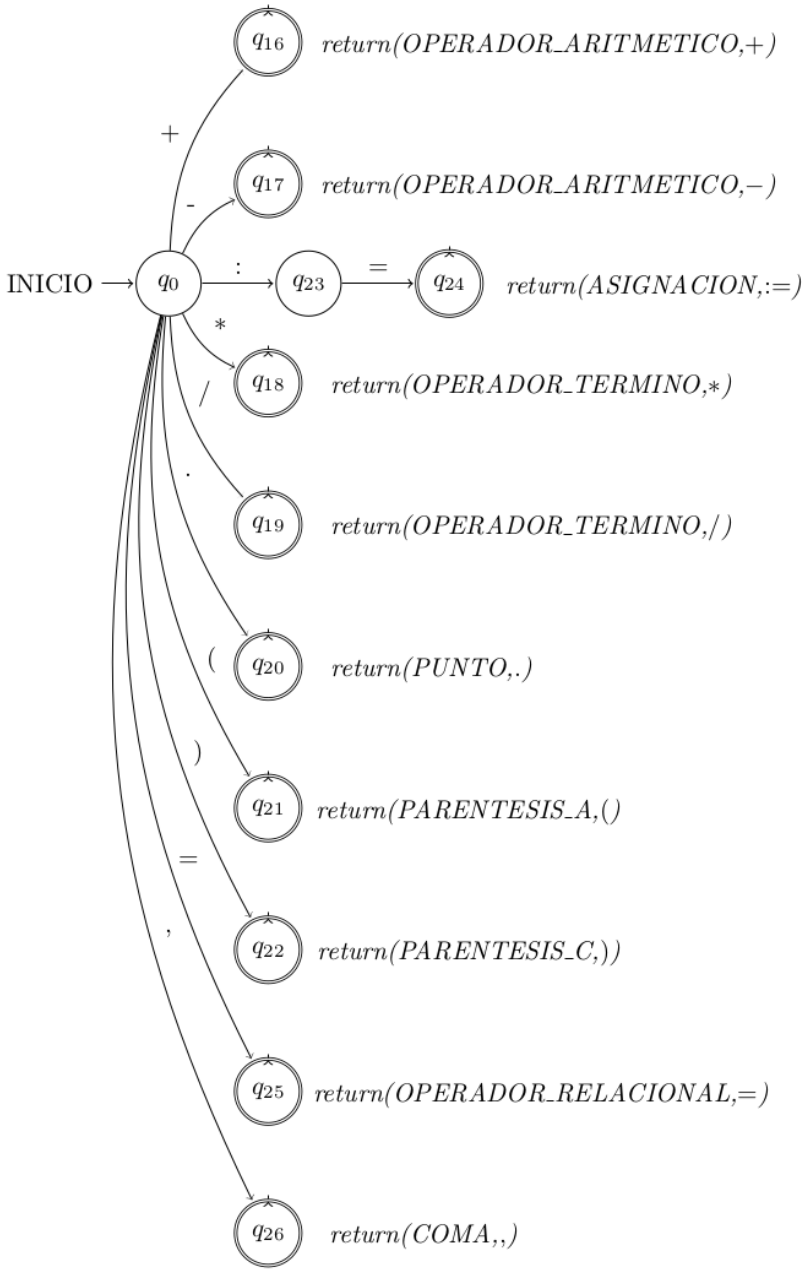

Fig. 2.2: Continuación de la Máquina de Estados reconocedora de tokens

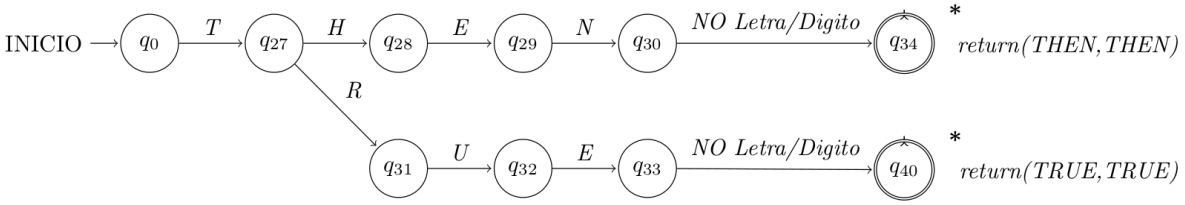

Fig. 2.3: *Máquina de Estados* para las *Palabras reservadas* del Lenguaje.

De forma análoga a la representada en la figura 2.3 están definidas todas las palabras que conforman el subconjunto reducido de *Pascal*. Por cuestiones de presentación se deja únicamente este ejemplo para la comprensión general.

Éstas son: **and begin do else end false function if integer procedure program read then true var while write**.

APLICATIVO

En todo diseño y desarrollo de un compilador, el resultado final consta, en última instancia, de un producto de Software listo para su utilización, que será desarrollado tanto en esta sección, como en capítulos posteriores.

En esta sección se referirá a todo lo relativo al aplicativo del **Analizador Léxico**, desde su utilización y especificación, así como casos de usos, preguntas frecuentes y documentación apropiada del mismo.

Para el desarrollo en general, tanto del *Analizador Léxico* como de los posteriores análisis (*Analizador Sintáctico Descendente Predictivo Recursivo* y *Análisis Semántico*), se realizó una implementación en lenguaje de programación *Python* y, como metodología, la maximización del atributo de calidad de legibilidad del software por sobre la optimalidad de ejecución o la simplificación del código en una arquitectura *Singleton*.

Al tratarse como un producto destinado al aprendizaje y con un objetivo netamente educativo, se desea que el código fuente sea fácilmente legible, optando facilitar el análisis de lo realizado por sobre el nivel de optimización logrado.

De la misma manera el desarrollo se enfocó en lograr una autodocumentación que facilite su análisis y estudio.

Este mismo enfoque se presenta durante todos los capítulos en lo referido al aplicativo.

A continuación se detalla el *Manual de Uso* del Aplicativo, la *Especificación de tratamiento de errores* en un nivel muy simple y finalmente *Casos de Uso* y ejemplos para ejecutarlo.

Manual de Uso

El analizador fue desarrollado íntegramente en lenguaje de programación *Python*, por lo que se requiere que el sistema operativo objetivo tenga instalado *Python 2.7*.[1] En el caso de que este corriendo un sistema operativo que no tenga *Python* instalado, referimos al siguiente enlace de instalación oficial `https://www.python.org/downloads/`

[1] Los sistemas GNU/Linux en su mayoría incorporan por defecto en su instalación básica estas librerías

Para su utilización, se distribuye el aplicativo en ejecutables *multi-arquitectura* para sistemas operativos GNU/Linux.

Existen actualmente dos posibles ejecuciones del *Analizador Léxico*: ejecución de forma independiente o ejecución en conjunto.

Si se opta por la ejecución de forma independiente, basta con ingresar en la carpeta donde se encuentre el aplicativo y ejecutarlo por Terminal de la siguiente forma:

```
$ ./analizadorLexico.py -s archivoDePrueba.ext
```

Fig. 2.4: $ simboliza el puntero del interprete de la terminal, '-s' indica la ejecución *standalone* y archivoDePrueba.ext es el fichero a analizar

Como puede observarse en la figura 2.4 se requiere indicar mediante la opción '-s' que se espera la ejecución en modo '*Stand-Alone*' o *Ejecución Independiente* del Analizador Léxico. En este modo salva los resultados en un archivo de extensión *'.tokens'* con el mismo nombre que el archivo recibido por parámetro con el objetivo de verificar su correcto funcionamiento.

Asimismo existen otras posibilidades de ejecución, mediante *flags* o parametros opcionales que permiten la ejecución *verbosa* del aplicativo. Ésta incluye todas las impresiones de *debugging* y salidas de control. Para ello basta con añadir el parámetro **-v** como muestra la figura 2.5

```
$ ./analizadorLexico -s -v archivoDePrueba.ext
```

Fig. 2.5: $ simboliza el puntero del interprete de la terminal, -v indica *ejecución verbosa* y archivoDePrueba.ext es el fichero a analizar

El resultado de la ejecución del aplicativo puede observarse siempre por salida estándar de la *Terminal* o *Consola*.

La ejecución verbosa permite comprender el funcionamiento interno del aplicativo, por lo que si se analiza su salida detenidamente y con la documentación provista por este informe (*máquina de estados finitos* por ejemplo) resulta sencillo realizar un seguimiento de la traza de ejecución.

Es posible que requiera permisos de ejecución para poder utilizar el aplicativo. Si falla su ejecución por estas causas, para solucionarlo basta con ejecutar los siguientes comandos:

```
$ chmod +x analizadorLexico.py
```

Fig. 2.6: Brindar permisos de ejecución al aplicativo por Terminal

Y luego continuar con los pasos descriptos anteriormente.

Errores

En este punto del desarrollo del compilador, el enfoque de detección de errores y rutinas de recuperación esta limitado, esencialmente, al nivel del analizador léxico. Esto permite únicamente encontrar errores básicos sobre **comentarios que no cierran** y **símbolos no legales**.

Para el primero de los errores, el aplicativo toma como política, **recuperación en modo pánico**, omitiendo todo símbolo que aparezca luego de la apertura de un comentario en búsqueda del cierre del comentario. En caso de que el fichero termine antes de que se encuentre el cierre del comentario, emite un aviso de error adecuado.

Para el error de **símbolos no legales** dentro del archivo a analizar, el aplicativo emite un error al finalizar el análisis informando que existe un **token no reconocido** en la linea de código específica.

Todos los errores encontrados en el análisis léxico se muestran por salida estándar en la terminal donde se ejecuta el aplicativo, de la forma: *[numero de linea]*: **Error**.

Ejemplo de Utilización

A continuación se adjuntan ejemplos de ficheros que pueden ser utilizados para probar el aplicativo junto con la resolución del analizador -si existen errores -.

```
1 PROGRAM prueba
2 function funcion ()
3 var
4 {Comentario seguramente util y perspicaz}
5 integer a
6     begin
7       a := 10
8     end
9 var integer b integer a
10 BEGIN
11     b:=1000
12     false
13     b and false
14     if(a > b) THEN
15         begin
16           true
17         end
18     else
19         noidentado
20 END.
```

```
Analisis Finalizado. No hay errores detectados
```

Fig. 2.7: Impresión de errores

```
1 PROGRAM prueba
2 function funcion ()
3 var
4 {Comentario seguramente util y perspicaz}
5 integer a
6 begin
7   a := 10
8 end
9
10 var integer b integer a
11 BEGIN
12     b:=1000
13     false
14     b and false
15     if(a > b) THEN
16         begin
17             {{{{Como todo programador
       comentario que no cierra en algun
       punto de su vida
18           true
19         end
20     else
21         noidentado
22 END.
```

salida estándar

```
ERRORES DETECTADOS: 1
[EOF] Final de archivo inesperado: Comentario no finalizado
```

Fig. 2.8: Impresión de errores

```
1 PROGRAM prueba
2 function funcion ()
3 var
4 {Comentario seguramente util y perspicaz}
5 integer a
6 begin
7   a := 10
8 end
9
10 var integer b integer a
11 BEGIN
12     b:=1000
13     false
14     b and false
15     if(a > b) THEN
16         begin
17           {simbolo no perteneciente al
       lenguaje}
18             a := @
19             true
20         end
21     else
22         noidentado
23 END.
```

salida estándar

```
ERRORES DETECTADOS: 1
[21] Caracter(es) no reconocido(s) @
```

Fig. 2.9: Impresión de errores

```
1 PROGRAM prueba
2 function funcion ()
3 var
4 {Comentario seguramente util y perspicaz}
5 integer a
6 begin
7   a := 10
8 end
9
10 var integer b integer a
11 BEGIN
12     b:=1000
13     false
14     b and false
15     if(a > b) THEN
16         begin
17           {simbolo no perteneciente al
    lenguaje}
18           a := @
19           true
20           a := #ñ~
21           {comentario abierto
22             end
23     else
24         noidentado
25 END.
```

salida estándar

```
ERRORES DETECTADOS: 6
[21] Caracter(es) no reconocido(s) @
[23] Caracter(es) no reconocido(s) #
[23] Caracter(es) no reconocido(s) ñ
[23] Caracter(es) no reconocido(s) ñ
[23] Caracter(es) no reconocido(s) ~
[EOF] Final de archivo inesperado: Comentario no finalizado
```

Fig. 2.10: Impresión de errores

CONCLUSIONES

En este capítulo se ha determinado y especificado el *analizador léxico* encargado de identificar lexemas y componentes léxicos (*tokens*). Con esta finalidad se diseño una máquina de estados que representa de forma teórica el analizador.

También se ha logrado con éxito la implementación, en el aplicativo, de la misma máquina de estados, encargada de procesar el archivo fuente en *lenguaje Pascal*. Esta implementación permite tanto la ejecución continua del análisis como una ejecución linea por linea bajo demanda. Facilitando la implementación del posterior *analizador sintáctico*.

Se puede afirmar que se cumplió con los objetivos principales propuestos para esta etapa del compilador y la implementación lograda es suficiente para comenzar con la siguiente etapa prevista: el *Analizador Sintáctico*.

Capítulo 3

Analizador Sintáctico Descendente Recursivo Predictivo

El **Análisis Descendente Recursivo Predictivo** es una forma especial de análisis descendente recursivo donde el símbolo de preanálisis (lookahead) determina unívocamente (sin ambigüedad) el procedimiento seleccionado para cada no terminal.

GRAMÁTICA MODIFICADA

EBNF

Eliminar clausura de Kleene en <programa>

⟨*programa*⟩ ::= 'Program' ⟨*identificador*⟩ ⟨*declaracionVariables*⟩**?** (⟨*declaracionPyF*⟩)*****
'begin' ⟨*sentenciaCompuesta*⟩***** 'end.'

Sin clausura de Kleene

⟨*programa*⟩ ::= 'Program' ⟨*identificador*⟩ ⟨*declaracionVariableOpt*⟩ ⟨*declaracionPyFRep*⟩
'begin' ⟨*programaRepSentencia*⟩ 'end.'

⟨*declaracionVariableOpt*⟩ ::= ⟨*declaracionVariables*⟩
| λ

⟨*programaRepSentencia*⟩ ::= ⟨*compuesta*⟩ ⟨*programaRepSentencia*⟩
| λ

⟨*declaracionPyFRep*⟩ ::= ⟨*declaracionPyF*⟩ ⟨*declaracionPyFRep*⟩
| λ

Eliminar clausura de Kleene en <identificador>

⟨*identificador*⟩ ::= ⟨*letra*⟩$^{+}$ (⟨*digito*⟩* ⟨*letra*⟩*)*

Sin clausura de Kleene

⟨*identificador*⟩ ::= ⟨*letra*⟩ ⟨*identificadorRep*⟩

⟨*identificadorRep*⟩ ::= ⟨*letra*⟩ ⟨*identificadoRep*⟩
| ⟨*digito*⟩ ⟨*identificadorRep*⟩
| λ

Eliminar clausura de Kleene en <declaracionVariables>

⟨*declaracionVariables*⟩ ::= 'var' **(** ⟨*listaVariables*⟩**)**$^{+}$

Sin clausura de Kleene

⟨*declaracionVariables*⟩ ::= 'var' ⟨*listaVariables*⟩ ⟨*listaVariablesRep*⟩

⟨*listaVariablesRep*⟩ ::= ⟨*listaVariables*⟩
| λ

Eliminar clausura de Kleene en <listaIdentificador>

⟨*listaIdentificador*⟩ ::= ⟨*identificador*⟩ **(**, ⟨*identificador*⟩**)***

Sin clausura de Kleene

⟨*listaIdentificador*⟩ ::= ⟨*identificador*⟩ ⟨*listaIdentificadorRep*⟩

⟨*listaIdentificadorRep*⟩ ::= ' , ' ⟨*identificador*⟩ ⟨*listaIdentificadorRep*⟩
| λ

Eliminar clausura de Kleene en <declaracionPyf>

⟨*declaracionPyf*⟩ ::= 'procedure' ⟨*identificador*⟩ **(** ⟨*parametrosFormales*⟩**)*** ' ; '
⟨*declaracionVariables*⟩**?** **(**⟨*declaracionPyf*⟩**)*** ⟨*sentenciaCompuesta*⟩
| 'function' ⟨*identificador*⟩ ⟨*parametrosFormales*⟩* ' : '
⟨*tipoVariables*⟩' ; ' ⟨*declaracionVariables*⟩**?** **(**⟨*declaracionPyf*⟩**)***
⟨*sentenciaCompuesta*⟩

Sin clausura de Kleene

⟨*declaracionPyf*⟩ ::= ‘`procedure`’ ⟨*identificador*⟩ ⟨*parametrosFormales*⟩ ‘`;`’
⟨*declaracionVariableOpt*⟩ ⟨*declaracionPyfRep*⟩ ⟨*sentenciaCompuesta*⟩
| ‘`function`’ ⟨*identificador*⟩ ⟨*parametrosFormales*⟩ ‘`:`’ ⟨*tipoVariables*⟩
‘`;`’ ⟨*declaracionVariableOpt*⟩ ⟨*declaracionPyfRep*⟩ ⟨*sentenciaCompuesta*⟩

⟨*declaracionVariableOpt*⟩ ::= ⟨*declaracionVariables*⟩
| λ

Eliminar clausura de Kleene en <parametrosFormales>

⟨*parametrosFormales*⟩ ::= ‘`(`’ ⟨*listaIdentificador*⟩ ‘`:`’ ⟨*tipoVariables*⟩ **(**‘`;`’ ⟨*ParametrosFormales*⟩**)***

Sin clausura de Kleene

⟨*parametrosFormales*⟩ ::= ‘`(`’⟨*listaIdentificador*⟩ ‘`:`’ ⟨*tipoVariables*⟩ ⟨*parametrosFormalesRep*⟩
‘`)`’
| λ

⟨*parametrosFormalesRep*⟩ ::= ‘`;`’ ⟨*listaIdentificador*⟩ ‘`:`’ ⟨*tipoVariables*⟩
| λ

Eliminar clausura de Kleene en <sentenciaCompuesta> y <compuesta>

⟨*sentenciaCompuesta*⟩ ::= ‘`begin`’ **(** ⟨*compuesta*⟩ **)*** ‘`end`’ ‘`;`’

⟨*compuesta*⟩ ::= ⟨*sentencia*⟩ **(** ‘`;`’⟨*compuesta*⟩ **)***
| **(**⟨*sentenciaCompuesta*⟩**)***

Sin clausura de Kleene y restructuración de la gramatica

⟨*sentenciaCompuesta*⟩ ::= ‘`begin`’ ⟨*compuesta*⟩ ‘`end`’

⟨*compuesta*⟩ ::= ⟨*sentencia*⟩ ⟨*sentenciaOptativa*⟩
| ⟨*sentenciaCompuesta*⟩
| λ

⟨*sentenciaOptativa*⟩ ::= ‘`;`’ ⟨*compuesta*⟩
| λ

Eliminar clausura de Kleene en <sentencia>, <llamadaProcedimiento> y <asignacion>

⟨*sentencia*⟩ ::= ⟨*ifthen*⟩
| ⟨*mientras*⟩
| ⟨*identificador*⟩ ⟨*asignacion*⟩
| ⟨*identificador*⟩**?** ⟨*llamadaProcedimiento*⟩

⟨*llamadaProcedimiento*⟩ ::= ‘’(⟨*expresionGeneral*⟩ **(**,⟨*parametrosReales*⟩**)*‘)’?**
| ‘`write(`’⟨*expresionGeneral*⟩ ‘`)`’
| ‘`read(`’ ⟨*identificador*⟩ ‘`)`’

⟨*asignacion*⟩ ::= ‘`:=`’ ⟨*expresionGeneral*⟩

Sin clausura de Kleene y restructuración de la gramatica

⟨*sentencia*⟩ ::= ⟨*ifthen*⟩
| ⟨*mientras*⟩
| ⟨*identificador*⟩ ⟨*asignacionollamada*⟩
| ‘`write(`’⟨*expresionGeneral*⟩ ‘`)`’
| ‘`read(`’ ⟨*identificador*⟩ ‘`)`’

⟨*asignacionollamada*⟩ ::= ‘`:=`’ ⟨*expresionGeneral*⟩
| ⟨*llamada*⟩
| λ

Factorizar a izquierda <ifthen>

⟨*ifthen*⟩ ::= ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ⟨*compuesta*⟩ ‘`end`’ ‘`;`’
| ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ⟨*compuesta*⟩ ‘`end`’‘`else`’ ⟨*compuesta*⟩
| ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ‘`begin`’ ⟨*compuesta*⟩ ‘`end`’ ‘`;`’
| ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ‘`begin`’ ⟨*compuesta*⟩ ‘`end`’ ‘`else`’ ⟨*compuesta*⟩

factorización a izquierda

⟨*ifthen*⟩ ::= ‘`if`’ ⟨*expresionGeneral*⟩ ‘`then`’ ⟨$ifthen_1$⟩

⟨$ifthen_1$⟩ ::= ⟨*sentenciaCompuesta*⟩ ⟨*alternativa*⟩
| ⟨*sentencia*⟩ ⟨*alternativa*⟩

⟨*alternativa*⟩ ::= ‘`;`’
| ‘`else`’ ⟨*compuesta*⟩

Factorizar a izquierda <mientras>

⟨*mientras*⟩ ::= ‘`while`’ ⟨*expresionGeneral*⟩ ‘`do`’ ‘`begin`’ ⟨*compuesta*⟩ ‘`end`’
| ‘`while`’ ⟨*expresionGeneral*⟩ ‘`do`’ ⟨*sentencia*⟩

factorización a izquierda

⟨*mientras*⟩ ::= ‘`while`’ ⟨*expresionGeneral*⟩ ‘`do`’ ⟨$mientras_1$⟩

⟨$mientras_1$⟩ ::= ⟨*sentenciaCompuesta*⟩
| ⟨*sentencia*⟩

Quitar recursión a izquierda <expresionAritmetica>

⟨*expresionAritmetica*⟩ ::= ⟨*expresionAritmetica*⟩ ‘+’ ⟨*termino*⟩
| ⟨*expresionAritmetica*⟩ ‘-’ ⟨*termino*⟩
| ⟨*termino*⟩

Sin recursion a izquierda

⟨*expresionAritmetica*⟩ ::= ⟨*termino*⟩ ⟨$expresionAritmetica_1$⟩

⟨$expresionAritmetica_1$⟩ ::= ‘+’ ⟨*termino*⟩ ⟨$expresionAritmetica_1$⟩
| ‘-’ ⟨*termino*⟩ ⟨$expresionAritmetica_1$⟩
| λ

Quitar recursión a izquierda <termino>

⟨*termino*⟩ ::= ⟨*termino*⟩ ‘*’ ⟨*factor*⟩
| ⟨*termino*⟩ ‘/’ ⟨*factor*⟩
| ⟨*factor*⟩

Sin recursion a izquierda

⟨*termino*⟩ ::= ⟨*factor*⟩ ⟨$termino_1$⟩

⟨$termino_1$⟩ ::= ‘*’ ⟨*factor*⟩ ⟨$termino_1$⟩
| ‘/’ ⟨*factor*⟩ ⟨$termino_1$⟩
| λ

Factorizar a izquierda <factor> y reestructuración

⟨*factor*⟩ ::= ⟨*identificador*⟩
| ⟨*llamadaProcedimiento*⟩
| ⟨*digitos*⟩
| ‘`true`’
| ‘`false`’
| ‘(’⟨*expresionGeneral*⟩‘)’

Sin recursion a izquierda

⟨*factor*⟩ ::= ⟨*identificador*⟩ ⟨*llamada*⟩
| ‘`write`’‘(’‘<expresionGeneral> ‘)’
| ‘`read`’‘(’ ⟨*identificacación*⟩‘)’
| ⟨*digitos*⟩
| ‘`true`’
| ‘`false`’
| ‘(’⟨*expresionGeneral*⟩‘)’

⟨*llamada*⟩ ::= ‘(’⟨*expresionGeneral*⟩⟨*parametrosReales*⟩‘)’
| λ

Eliminar clausura de Kleene en <digitos>

⟨*digitos*⟩ ::= ⟨*digito*⟩$^+$
| - ⟨*digito*⟩$^+$
| + ⟨*digito*⟩$^+$

Sin clausura de Kleene

⟨*digitos*⟩ ::= ⟨*digito*⟩⟨*digitosRep*⟩
| ‘-’ ⟨*digito*⟩⟨*digitosRep*⟩
| ‘+’ ⟨*digito*⟩⟨*digitosRep*⟩

⟨*digitosRep*⟩ ::= ⟨*digito*⟩ ⟨*digitosRep*⟩
| λ

Eliminar clausura de Kleene en <parametrosReales>

⟨*parametrosReales*⟩ ::= (‘,’⟨*expresionGeneral*⟩)*

Sin clausura de Kleene y reestructuración de la gramática

⟨*parametrosReales*⟩ ::= ‘,’ ⟨*expresionGeneral*⟩ ⟨*parametrosReales*⟩
| λ

Quitar recursión a izquierda <expresionGeneral> y eliminasion de <expresion>

⟨*expresionGeneral*⟩ ::= ⟨*expresionGeneral*⟩ ‘or’ ⟨*compararAnd*⟩
| ⟨*compararAnd*⟩

⟨*compararAnd*⟩ ::= ⟨*compararAnd*⟩ ‘and’ ⟨*expresion*⟩
| ⟨*expresion*⟩

⟨*expresion*⟩ ::= ‘not’**?**⟨*expresionAritmetica*⟩**(**⟨*operadorRelacional*⟩⟨*expresionAritmetica*⟩**)?**

Sin recursion a izquierda

⟨*expresionGeneral*⟩ ::= ⟨*not*⟩ ⟨*expresionAritmetica*⟩⟨*expresionRelacional*⟩⟨*expresionAritmetica*⟩⟨*compara*

⟨*expresionGeneral*$_1$⟩ ::= ‘or’ ⟨*expresionGeneral*⟩
| λ

⟨*compararAnd*⟩ ::= ‘and’ ⟨*not*⟩ ⟨*expresionAritmetica*⟩⟨*expresionRelacional*⟩⟨*expresionAritmetica*⟩⟨*con*
| λ

⟨*not*⟩ ::= ‘not’
| λ

⟨*expresionRelacional*⟩ ::= ⟨*operadorRelacional*⟩ ⟨*expresionAritmetica*⟩
| λ

REGLAS DE PRODUCCIÓN PARA GRAMÁTICA LL1

⟨*programa*⟩ ::= ‘Program’ ⟨*identificador*⟩ ‘;’ ⟨*declaracionVariableOpt*⟩ ⟨*declaracionPyFRep*⟩
‘begin’ ⟨*programaRepSentencia*⟩ ‘end.’

⟨*identificador*⟩ ::= ⟨*letra*⟩ ⟨*identificadorRep*⟩

⟨*identificadorRep*⟩ ::= ⟨*letra*⟩ ⟨*identificadoRep*⟩
| ⟨*digito*⟩ ⟨*identificadorRep*⟩
| λ

⟨*programaRepSentencia*⟩ ::= ⟨*compuesta*⟩ ⟨*programaRepSentencia*⟩
| λ

⟨*declaracionVariableOpt*⟩ ::= ⟨*declaracionVariables*⟩
| λ

⟨*declaracionVariables*⟩ ::= '`var`' ⟨*listaVariables*⟩ ⟨*listaVariablesRep*⟩

⟨*listaVariables*⟩ ::= ⟨*listaIdentificador*⟩ '`:`' ⟨*tipoVariable*⟩ '`;`'

⟨*tipoVariables*⟩ ::= '`integer`'
| '`boolean`'

⟨*listaVariablesRep*⟩ ::= ⟨*listaVariables*⟩
| λ

⟨*listaIdentificador*⟩ ::= ⟨*identificador*⟩ ⟨*listaIdentificadorRep*⟩

⟨*listaIdentificadorRep*⟩ ::= '`,`' ⟨*identificador*⟩ ⟨*listaIdentificadorRep*⟩
| λ

⟨*declaracionPyf*⟩ ::= '`procedure`' ⟨*identificador*⟩ ⟨*parametrosFormales*⟩'`;`' ⟨*declaracionVariableOpt*⟩
⟨*declaracionPyfRep*⟩ ⟨*sentenciaCompuesta*⟩
| '`function`' ⟨*identificador*⟩ ⟨*parametrosFormales*⟩ '`:`' ⟨*tipoVariables*⟩
'`;`' ⟨*declaracionVariableOpt*⟩ ⟨*declaracionPyfRep*⟩ ⟨*sentenciaCompuesta*⟩

⟨*parametrosFormales*⟩ ::= '`(`'⟨*listaIdentificador*⟩ '`:`' ⟨*tipoVariable*⟩ ⟨*parametrosFormalesRep*⟩'`)`'
| λ

⟨*parametrosFormalesRep*⟩ ::= '`;`' ⟨*listaIdentificador*⟩ '`:`' ⟨*tipoVariable*⟩ ⟨*parametrosFormalesRep*⟩
| λ

⟨*declaracionPyfRep*⟩ ::= ⟨*declaracionPyf*⟩ ⟨*declaracionPyfRep*⟩
| λ

⟨*sentenciaCompuesta*⟩ ::= '`begin`' ⟨*compuesta*⟩ '`end`'

⟨*compuesta*⟩ ::= ⟨*sentencia*⟩ ⟨*sentenciaOptativa*⟩
| ⟨*sentenciaCompuesta*⟩
| λ

⟨*sentenciaOptativa*⟩ ::= '`;`' ⟨*compuesta*⟩
| λ

⟨*sentencia*⟩ ::= ⟨*ifthen*⟩
| ⟨*mientras*⟩
| ⟨*identificador*⟩ ⟨*asignacionollamada*⟩
| '`write`' '`(`'⟨*expesionGeneral*⟩ '`)`'
| '`read(`' ⟨*identificador*⟩ '`)`'

⟨*ifthen*⟩ ::= '`if`' ⟨*expresionGeneral*⟩ '`then`' ⟨$ifthen_1$⟩

⟨$ifthen_1$⟩ ::= ⟨*sentenciaCompuesta*⟩ ⟨*alternativa*⟩
| ⟨*sentencia*⟩ ⟨*alternativa*⟩

⟨*alternativa*⟩ ::= '`;`'
| '`else`' ⟨*compuesta*⟩

⟨*mientras*⟩ ::= '`while`' ⟨*expresionGeneral*⟩ '`do`' ⟨*mientras*⟩

⟨*mientras1*⟩ ::= ⟨*sentenciaCompuesta*⟩
| ⟨*sentencia*⟩

⟨*asignacionollamada*⟩ ::= '`:=`' ⟨*expresionGeneral*⟩
| ⟨*llamada*⟩
| λ

⟨*expresionAritmetica*⟩ ::= ⟨*termino*⟩ ⟨$expresionAritmetica_1$⟩

⟨*expresionAritmetica*$_1$⟩ ::= '+' ⟨*termino*⟩ ⟨*expresionAritmetica*$_1$⟩
| '-' ⟨*termino*⟩ ⟨*expresionAritmetica*$_1$⟩
| λ

⟨*termino*⟩ ::= ⟨*factor*⟩ ⟨*termino*$_1$⟩

⟨*termino*$_1$⟩ ::= '*' ⟨*factor*⟩ ⟨*termino*$_1$⟩
| '/' ⟨*factor*⟩ ⟨*termino*$_1$⟩
| λ

⟨*factor*⟩ ::= ⟨*identificador*⟩ ⟨*llamada*⟩
| 'write' '('⟨*expresionGeneral*⟩')'
| 'read' '(' ⟨*identificador*⟩ ')'
| ⟨*digitos*⟩
| 'true'
| 'false'
| '(' ⟨*expresionGeneral*⟩ ')'

⟨*digitos*⟩ ::= ⟨*digito*⟩ ⟨*digitosRep*⟩
| '-' ⟨*digito*⟩⟨*digitosRep*⟩
| '+' ⟨*digito*⟩⟨*digitosRep*⟩

⟨*digitosRep*⟩ ::= ⟨*digito*⟩ ⟨*digitosRep*⟩
| λ

⟨*llamada*⟩ ::= '(' ⟨*expresionGeneral*⟩ ⟨*parametrosReales*⟩ ')'
| λ

⟨*parametrosReales*⟩ ::= ',' ⟨*expresionGeneral*⟩ ⟨*parametrosReales*⟩
| λ

⟨*expresionGeneral*⟩ ::= ⟨*not*⟩ ⟨*expresionAritmetica*⟩ ⟨*expresionRelacional*⟩ ⟨*compararAnd*⟩
⟨*expresionGeneral*$_1$⟩

⟨*expresionGeneral*$_1$⟩ ::= 'or' ⟨*expresionGeneral*⟩
| λ

⟨*not*⟩ ::= 'not'
| λ

⟨*expresionRelacional*⟩ ::= ⟨*operadorRelacional*⟩ ⟨*expresionAritmetica*⟩
| λ

⟨*compararAnd*⟩ ::= 'and' ⟨*not*⟩ ⟨*expresionAritmetica*⟩ ⟨*expresionRelacional*⟩ ⟨*compararAnd*⟩
| λ

⟨*operadorRelacional*⟩ ::= '=' | '<>' | '<' | '<=' | '>' | '>='

⟨*letra*⟩ ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '_'

⟨*digito*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

DEMOSTRACIÓN

Definiciones previas

Para poder afirmar que la gramática dada anteriormente es de tipo *LL1*, es necesario demostrar que cumple con la definición apropiada:

Una gramática cuya tabla de análisis no muestra entradas múltiples es llamada gramática LL(1).

Las gramáticas *LL(1)* tienen propiedades interesantes:

- No ambiguas
- No recursivas a izquierda
- Factorizadas a izquierda

Para la construcción de la tabla, es necesario calcular dos funciones para cada elemento que pertenece a *Símbolos Terminales* ∪ *Símbolos No Terminales*, las funciones *Primeros(x)* y *Siguientes(x)*.

Ambas se definen a continuación:

Función *Primeros(X)*

Para todo símbolo (terminal o no-terminal) de la gramática, se cumple que:

- Si X es un **terminal**, entonces $Primeros(X) = X$
- Si $X \rightarrow \lambda$ es una producción, entonces añadir λ a $Primeros(X)$
- Si X es un **no-terminal** y $X \rightarrow Y_1 Y_2 ... Y_k$, entonces añadir "$a$" en $Primeros(X)$ si "a" está en $Primeros(Y_i)$ y λ está en $Primeros(Y_1), ..., Primeros(Y_{i-1})$

Para calcular la función *Primeros* de una cadena, del tipo $Primeros(X_1 X_2 ... X_n)$: Incluimos el valor de $Primeros(X_1) - \{\lambda\}$ si λ pertenece a $Primeros(X_1)$, incluimos a $Primeros(X_2) - \{\lambda\}$ si λ pertenece a $Primeros(X_2)$, incluimos a $Primeros(X_3) - \{\lambda\}$ y así sucesivamente.

Función *Siguientes(N)*

Para cada *no-terminal N* Incluir $ en *Siguientes(S)* donde *S* es el símbolo inicial y $ el símbolo de fin de cadena.

Si existe una producción de la forma $A \rightarrow \alpha B \beta$ entonces incluir *Primeros*(β), excepto λ, en *Siguientes(B)*.

Si existe una producción de la forma $A \rightarrow \alpha B$ o $A \rightarrow \alpha B \beta$, donde *Primeros*($\beta$) contiene a λ, todo lo que esté en *Siguientes(A)* se incluye en *Siguientes(B)*.

Construcción de la tabla de análisis

Entrada: Una gramática G

Salida: La tabla de Análisis Sintáctico M

Método: Para cada producción $A \rightarrow \alpha$ seguir los dos pasos siguientes:

- para cada terminal a de *Primeros*(α) añadir $A \rightarrow \alpha$ a $M[A, a]$
- si λ está en *Primeros*(α) añadir $A \rightarrow \alpha$ a $M[A, b]$ para cada terminal b de *Siguientes(A)*
- si λ está en *Primeros*(α) y $ está en *Siguientes(A)* añadir $A \rightarrow \alpha$ a $M[A, \$]$

Aclaración: (cada entrada no definida de M es una entrada de “error”)

Para poder simplificar el análisis de la tabla lograda, cada entrada de "*error*" fue omitida, dejando solo las entradas que efectivamente tienen una transición asociada.

Con esto, bastaría calcular las funciones *Primero(x)* y *Siguiente(x)* para cada símbolo. Armar la tabla de Análisis asociada correspondiente donde se debe verificar que para cada cruce de la tabla exista una única regla de producción asociada.

Por ello, se simplificó la escritura para que la "Tabla" sea leída como pares ordenados de **no-terminal-terminal** que mapean -como si fuesen una función- a una única regla de producción escrita de forma análoga a su definición en la gramática.

ANALIZADOR SINTÁCTICO

En esta sección se presentara un *pseudo-código* que representa el diseño del *Analizador sintáctico descendente recursivo predictivo* estructurado de la siguiente manera:

- Regla de producción
- Pseudo-código asociado a la regla

⟨*programa*⟩ ::= 'Program' ⟨*identificador*⟩ ';' ⟨*declaracionVariableOpt*⟩ ⟨*declaracionPyFRep*⟩ 'begin' ⟨*programaRepSentencia*⟩ 'end.'

```
programa():
    if ( preanalisis == "program") :
                        match("program")
                        identificador()
                        match("punto_coma")
                        declaracionVariableOpt()
                        declaracionPyFRep()
                        match("begin")
                        programaRepSentencia()
                        match("end")
                        match("punto")
        else:
                        reportar("Error de sintaxis")
```

⟨*identificador*⟩ ::= ⟨*letra*⟩ ⟨*identificadorRep*⟩

```
identificador():
    if(preanalisis == "identificador"):
                match("identificador")
        else:
                reportar("Error de sintaxis")
```

⟨*programaRepSentencia*⟩ ::= ⟨*compuesta*⟩ ⟨*programaRepSentencia*⟩
| λ

```
void programaRepSentencia() {
    if (preanalisis in {"begin","read", "write","while",(@"if" , ' ' identificador ' '@)}) :
                        compuesta()
                        programaRepSentencia()
```

⟨*declaracionVariableOpt*⟩ ::= ⟨*declaracionVariables*⟩
| λ

```
declaracionVariableOpt():
    if ( preanalisis == "var") :
                        declaracionVariables()
```

⟨*declaracionVariables*⟩ ::= 'var' ⟨*listaVariables*⟩ ⟨*listaVariablesRep*⟩

```
declaracionVariables():
        if(preanalisis == "var"):
                match("var")
                listaVariables()
                listaVariablesRep()

        else:
                reportar("Error de sintaxis")
```

⟨*listaVariables*⟩ ::= ⟨*listaIdentificador*⟩ : ⟨*tipoVariable*⟩ ';'

```
listaVariables():
        if(preanalisis == "identificador"):
                listaIdentificador()
                match("dos_puntos")
                tipoVariables()
                match("punto_coma")
        else:
                reportar("Error de Sintaxis:")
```

⟨*tipoVariables*⟩ ::= ‘integer’
| ‘boolean’

```
tipoVariables():
    if( preanalisis == “integer”):
            match(“integer”)
        elif( preanalisis == “boolean”):
            match(“boolean”)
        else:
            reportar(“Error de sintaxis”)
```

⟨*listaVariablesRep*⟩ ::= ⟨*listaVariables*⟩
| λ

```
listaVariables():
        if(preanalisis == “identificador”):
            listaVariables()
```

⟨*listaIdentificador*⟩ ::= ⟨*identificador*⟩ ⟨*listaIdentificadorRep*⟩

```
listaIdentificador():
    if(preanalisis == “identificador”):
            identificador()
            listaIdentificadorRep()
        else:
            reportar(“Error de sintaxis”)
```

⟨*listaIdentificadorRep*⟩ ::= ‘,’ ⟨*identificador*⟩ ⟨*listaIdentificadorRep*⟩
| λ

```
listaIdentificadorRep():
    if( preanalisis == “coma”):
            match(“coma”)
            identificador()
            listaIdentificadorRep()
```

⟨*declaracionPyf*⟩ ::= ‘`procedure`’ ⟨*identificador*⟩ ⟨*parametrosFormales*⟩ ‘;’ ⟨*declaracionVariableOpt*⟩
⟨*declaracionPyfRep*⟩ ⟨*sentenciaCompuesta*⟩
| ‘`function`’ ⟨*identificador*⟩ ⟨*parametrosFormales*⟩ ‘:’ ⟨*tipoVariables*⟩
‘;’ ⟨*declaracionVariableOpt*⟩ ⟨*declaracionPyfRep*⟩ ⟨*sentenciaCompuesta*⟩

```
declaracionPyf():
        if ( preanalisis == “procedure”) :
                        match(“procedure”)
                        identificador()
                        parametrosFormales()
                        match(“punto_coma”)
                        declaracionVariableOpt()
                        declaracionPyfRep()
                        sentenciaCompuesta()
        elif ( preanalisis == “function”) :
                        match(“function”)
                        identificador()
                        parametrosFormales()
                        match(“dos_punto”)
                        tipoVariables()
                        match(“punto_coma”)
                        declaracionVariableOpt()
                        declaracionPyfRep()
                        sentenciaCompuesta()
        else:
                        reportar(“Error de sintaxis”)
```

⟨*parametrosFormales*⟩ ::= ‘(’ ⟨*listaIdentificador*⟩ : ⟨*tipoVariables*⟩ ⟨*parametrosFormalesRep*⟩
‘)’
| λ

```
parametrosRep()
        if ( preanalisis ==“parentesis_a”)  :
                    match(“parentesis_a”)
                    listaIdentificador()
                    match(“dos_punto”)
                    tipoVariables()
                    parametrosFormalesRep()
            match(“parentesis_c”)
```

⟨*parametrosFormalesRep*⟩ ::= ';' ⟨*listaIdentificador*⟩ ':' ⟨*tipoVariables*⟩ ⟨*parametrosFormalesRep*⟩
| λ

```
parametrosFormalesRep():
        if (preanalisis == "punto_coma"):
                match("punto_coma")
                listaIdentificador()
                match("dos_punto")
                tipoVariables()
                parametrosFormalesRep()
```

⟨*declaracionPyfRep*⟩ ::= ⟨*declaracionPyf*⟩ ⟨*declaracionPyfRep*⟩
| λ

```
declaracionPyfRep():
        if (preanalisis == "function" or preanalisis == "procedure"):
                declaracionPyf()
                declaracionPyfRep()
```

⟨*sentenciaCompuesta*⟩ ::= 'begin' ⟨*compuesta*⟩ 'end'

```
sentenciaCompuesta():
        if(preanalisis == "begin"):
                match("begin")
                compuesta()
                match("end")
        %       match("punto_coma")
        else:
                reportar("Error de sintaxis")
```

⟨*compuesta*⟩ ::= ⟨*sentencia*⟩ ⟨*sentenciaOptativa*⟩
| ⟨*sentenciaCompuesta*⟩ ⟨*compuesta*⟩
| λ

```
compuesta():
    if (preanalisis in {"write","while","read","if","identificador"}):
                sentencia()
                sentenciaOptativa()
        elif (preanalisis == "begin"):
                sentenciaCompuesta()
                compuesta()
```

⟨*sentenciaOptativa*⟩ ::= ‘;’ ⟨*compuesta*⟩
| λ

```
sentenciaOptativa():
    if(preanalisis == “punto_coma”):
            match(“punto_coma”)
            compuesta()
```

⟨*sentencia*⟩ ::= ⟨*ifthen*⟩
| ⟨*mientras*⟩
| ⟨*identificador*⟩ ⟨*asignacionollamada*⟩
| ‘`write`’‘(’⟨*expresionGeneral*⟩ ‘)’
| ‘`read`’ ‘(’ ⟨*identificador*⟩ ‘)’

```
sentencia():
        if(preanalisis == “if”):
            ifthen()
        elif(preanalisis == “while”):
            mientras()
        elif(preanalisis == “identificador”):
            identificador()
            asignacionollamada()
        elif(preanalisis == “write”):
            match(‘‘write’’)
            match(“parentesis_a”)
            expresionGenereal()
            match(“parentesis_c”)
        elif(preanalisis == “read”):
            match(“read”)
            match(“parentesis_a”)
            identificador()
        match(“parentesis_c”)
        else:
            reportar(“Error de sintaxis”)
```

⟨*ifthen*⟩ ::= 'if' ⟨*expresionGeneral*⟩ 'then' ⟨$ifthen_1$⟩

```
ifthen()
        if ( preanalisis == "if") :
                        match("if")
                        expresionGeneral()
                        match("then")
                        ifthen1()
            else:
                        reportar("Error de sintaxis")
```

⟨$ifthen_1$⟩ ::= 'begin' ⟨*compuesta*⟩ 'end' ⟨*alternativa*⟩
| ⟨*sentencia*⟩ ⟨*alternativa*⟩

```
ifthen1()
        if ( preanalisis == "begin") :
                        match("begin")
                        compuesta()
                        match("end")
                        alternativa()
        elif (preanalisis in {"read", "write","while","if","identificador"}) :
                        sentencia()
                        alternativa()
            else:
                        reportar("Error de sintaxis")
```

⟨*alternativa*⟩ ::= ';'
| 'else' ⟨*compuesta*⟩

```
alternativa():
        if   (preanalisis == "punto_coma") :
                        match("punto_coma")
        elif (preanalisis == "else") :
                        match("else")
                        compuesta()
        else:
                        reportar("Error de sintaxis")
```

⟨*mientras*⟩ ::= ‘`while`’ ⟨*expresionGeneral*⟩ ‘`do`’ ⟨$mientras_1$⟩

```
mientras()
        if ( preanalisis == “while”) :
                        match(“while”)
                        expresionGeneral()
                        match(“do”)
                        mientras1()
           else:
                        reportar(“Error de sintaxis”)
```

⟨$mientras_1$⟩ ::= ⟨*sentenciaCompuesta*⟩
| ⟨*sentencia*⟩

```
mientras1()
        if ( preanalisis == “begin”) :
                      sentenciaCompuesta()
           elif(preanalisis in {“read”, “write”,“while”,“if”,“identificador”})
               sentencia()

           else:
                        reportar(“Error de sintaxis”)
```

⟨*asignacionollamada*⟩ ::= ‘:=’ ⟨*expresionGeneral*⟩ ‘;’
| ⟨*llamada*⟩
| λ

```
asignacionollamada():
        if (preanalisis == “asignacion”):
                match(“asignacion”)
                expresionGeneral()
        elif (preanalisis == “parentesis_a”):
                llamada()
```

⟨*expresionAritmetica*⟩ ::= ⟨*termino*⟩ ⟨$expresionAritmetica_1$⟩

```
expresionAritmetica():
        if (preanalisis in {“write”,“true”,“false”,“read”,“operador_aritmetico”,
    “parentesis_a”,“identificador”,“numero”}) :
                termino()
                expresionAritmetica1()
        else:
                reportar(“Error de sintaxis”)
```

⟨*expresionAritmetica*$_1$⟩ ::= ‘+’ ⟨*termino*⟩ ⟨*expresionAritmetica*$_1$⟩
| ‘-’ ⟨*termino*⟩ ⟨*expresionAritmetica*$_1$⟩
| λ

```
expresionAritmetica1():
        if ( preanalisis == “operador_aritmetico”):
          match(“operador_aritmetico”)
          termino()
          expresionAritmetica1()
```

⟨*termino*⟩ ::= ⟨*factor*⟩ ⟨*termino*$_1$⟩

```
termino():
        if(preanalisis in {“write”,“true”,“false”,“read”,“operador_aritmetico”,
    “parentesis_a”,“identificador”,“numero”}) :
                        factor()
                        termino1()
        else:
                        reportar(“Error de Sintaxis”)
```

⟨*termino*$_1$⟩ ::= ‘∗’ ⟨*factor*⟩ ⟨*termino*$_1$⟩
| ‘/’ ⟨*factor*⟩ ⟨*termino*$_1$⟩
| λ

```
termino1():
    if (preanalisis == “operador_termino”) :
                        match(“operador_termino”)
                        factor()
                        termino1()
```

⟨*factor*⟩ ::= ⟨*identificador*⟩ ⟨*llamada*⟩
| ‘`write`’ ‘(’ ⟨*expresionGeneral*⟩ ‘)’
| ‘`read`’ ‘(’ ⟨*identificador*⟩ ‘)’
| ⟨*digitos*⟩
| ‘`true`’
| ‘`false`’
| ‘(’⟨*expresionGeneral*⟩‘)’

```
factor():
        if (preanalisis == "identificador"):
                                identificador()
                                llamada()
        elif (preanalisis == "write") :
                                match(''write'')
                                match("parentesis_a")
                                expresionGeneral()
                                match("parentesis_c")
        elif (preanalisis == "read") :
                                match("read")
                                match("parentesis_a")
                                identificador()
                                match("parentesis_c")
        elif (preanalisis == "numero" or preanalisis == ''operador_aritmetico''@)) :
                                digitos()
        elif (preanalisis == "true") :
                                match("true")
        elif (preanalisis == "false") :
                                match("false")
        elif (preanalisis == (@"parentesis_a"@)) :
                                match((@"parentesis_a"@))
                                expresionGeneral()
                                match("parentesis_c")
        else:
                                reportar("Error de sintaxis")
```

⟨*digitos*⟩ ::= ⟨*digito*⟩ ⟨*digitosRep*⟩
| ‘-’ ⟨*digito*⟩ ⟨*digitosRep*⟩
| ‘+’ ⟨*digito*⟩ ⟨*digitosRep*⟩

```
digitos()
        if(preanalisis == “numero”):
                match(“numero”)
        elif (preanalisis == “operador_aritmetico”):
                match(“operador_aritmetico”)
                match(“numero”)
        else:
                reportar(“Error de sintaxis”)
```

⟨*llamada*⟩ ::= ‘(’ ⟨*expresionGeneral*⟩ ⟨*parametrosReales*⟩ ‘)’
| λ

```
llamada():
        if(preanalisis in {“parentesis_a”}):
                        match(“parentesis_a”)
                        expresionGeneral()
                        parametrosReales()
                        match(“parentesis_c”)
```

⟨*parametrosReales*⟩ ::= ‘,’ ⟨*expresionGeneral*⟩ ⟨*parametrosRealesRep*⟩
| λ

```
parametrosReales():
        if(preanalisis in {“coma”}):
                match(“coma”)
                        expresionGeneral()
                        parametrosRealesRep()
```

⟨*expresionGeneral*⟩ ::= ⟨*not*⟩ ⟨*expresionAritmetica*⟩ ⟨*expresionRelacional*⟩ ⟨*compararAnd*⟩ ⟨*expresionGeneral*$_1$⟩

```
expresionGeneral():
    if(preanalisis in {"false", "true", "parentesis_a", "operador_aritmetico", "write", "read",
    "not","identificador","numero"}):
                no()
                expresionAritmetica()
                expresionRelacional()
                compararAnd()
                expresionGeneral1()
        else:
                reportar("Error de sintaxis")
```

⟨*expresionGeneral*$_1$⟩ ::= 'or' ⟨*expresionGeneral*⟩
| λ

```
expresionGeneral1():
    if(preanalisis == "or"):
            match("or")
            expresionGeneral()
```

⟨*not*⟩ ::= 'not'
| λ

```
no():
        if(preanalisis == "not"):
                match("not")
```

⟨*expresionRelacional*⟩ ::= ⟨*operadorRelacional*⟩ ⟨*expresionAritmetica*⟩
| λ

```
expresionRelacional():
    if(preanalisis == "operador_relacional"):
                operadorRelacional()
                expresionAritmetica()
```

⟨*compararAnd*⟩ ::= 'and'⟨*not*⟩ ⟨*expresionAritmetica*⟩⟨*operadorRelacional*⟩ ⟨*compararAnd*⟩
| λ

```
1 compararAnd():
2     if(preanalisis == "and"):
3                 match("and")
4                 no()
5                 expresionAritmetica()
6                 operadorRelacional()
7                 compararAnd()
```

⟨*operadorRelacional*⟩ ::= '=' | '<>' | '<' | '<=' | '>' | '>='

```
1 operadorRelacional()
2         if ( preanalisis == "operador_relacional"):
3                         match("operador_relacional")
4         else:
5                         reportar("Error de sintaxis")
```

APLICATIVO

En esta sección se detalla todo lo relativo al producto final en cuestión, desde su utilización y especificación, así como casos de usos, preguntas frecuentes y documentación apropiada del mismo.

Como detalla el capítulo anterior, el criterio del aplicativo es educacional e incremental, por lo que toda especificación ya detallada, es conservada y aplicada.

Por la naturaleza propia del analizador sintáctico, su ejecución requiere del analizador léxico implementado anteriormente, por lo que se podrá observar en su ejecución detalles de lo descripto en la utilización del analizador léxico.

Cabe destacar que si bien los analizadores léxicos y sintácticos se encuentran en archivos diferenciados, se intercomunican a través de un protocolo de mensajes (particularmente, por medio de funciones encapsuladas en sus *packages* propios de *Python* y con parametrizaciones acordes) logrando una ejecución propia para un compilador de *una sola pasada*.

A continuación se detalla el *Manual de Uso* del Aplicativo, la *Especificación de tratamiento de errores* en un nivel muy simple y finalmente *Casos de Uso* y ejemplos para ejecutar el analizador sintáctico.

Manual de Uso

El analizador fue desarrollado íntegramente en lenguaje de programación *Python*, por lo que se requiere que el sistema operativo objetivo tenga instalado *Python 2.7*.[1]

Para su utilización, se distribuye el aplicativo en ejecutables *multi-arquitectura* para sistemas operativos GNU/Linux.

Para la ejecución del Analizador Sintáctico basta con ingresar en la carpeta donde se encuentre el aplicativo y ejecutarlo por Terminal de la siguiente forma:

```
1 $ ./aplicativo archivoDePrueba.ext
```

Fig. 3.1: $ simboliza el puntero del interprete de la terminal y archivoDePrueba.ext es el fichero a analizar

El resultado de la ejecución del aplicativo puede observarse siempre por salida estándar de la *Terminal* o *Consola*.

De forma análoga a lo detallado en el capítulo anterior, existen otras posibilidades de ejecución, mediante *flags* o parametros opcionales que permiten la ejecución *verbosa* del aplicativo. Ésta incluye todas las impresiones de *debugging* y salidas de control diferenciadas según el nivel de detalle requerido. Para ello basta con añadir el parámetro **-v** como muestra la figura 3.2

```
1 $ ./aplicativo -v archivoDePrueba.ext
```

Fig. 3.2: $ simboliza el puntero del interprete de la terminal, -v indica *ejecución verbosa de grado 1* y archivoDePrueba.ext es el fichero a analizar

El aplicativo cuenta con dos niveles de detalle para la salida *verbosa*. El primer nivel, donde únicamente se indica una bandera (‘*-v*’) para la impresión que detalla los mensajes del analizador Sintáctico y un segundo nivel de detalle, que se identifica con dos banderas (‘-vv’) donde además de lo detallado por el primer nivel, se añaden las impresiones del *Analizador Léxico*.

Es posible que requiera permisos de ejecución para poder utilizar el aplicativo. Si falla su ejecución por estas causas, para solucionarlo basta con ejecutar los siguientes comandos:

[1] Los sistemas GNU/Linux en su mayoría incorporan por defecto en su instalación básica estas librerías

```
1        $ chmod +x aplicativo
```

Fig. 3.3: Brindar permisos de ejecución al aplicativo por Terminal

Y luego continuar con los pasos descriptos anteriormente.

Errores

Para el objetivo de esta etapa, el enfoque de detección de errores y rutinas de recuperación esta limitado, esencialmente, al nivel del analizador léxico y sintáctico predictivo. En particular para el analizador sintáctico, permite únicamente encontrar errores básicos sobre **identificadores inválidos**.

Los **identificadores inválidos** no son analizados en profundidad en este nivel de desarrollo, ya que no corresponde al analizador sintáctico determinar si se trata de un error en la cadena, un número seguido de un identificador o un error de **"typo"**, por lo que se analizan de acuerdo a la natural correspondencia que tenga la cadena en su conformación (por ejemplo, en la cadena *"23ab"* si comienza con dígitos será tratada como un número y luego como un identificador a las letras que le siguen). Sin embargo se realiza un chequeo exhaustivo para evitar la definición de palabras reservadas como identificadores.

Si bien el manejo de errores es limitado, se optó como decisión de diseñó del analizador sintáctico, tratar los errores mediante "*modo pánico*": al encontrar una sentencia que contenga un error, se detiene el análisis de la misma y se avanza -descartando los *tokens*- hasta encontrar un *token* de sincronización o un delimitador. Luego se prosigue el análisis con el código restante.

Con la estrategia antes detallada, se espera maximizar la cantidad de errores detectables por ciclo de análisis, permitiendo una depuración mas eficiente para el usuario del compilador.

Todos los errores encontrados en el análisis se muestran por salida estándar en la terminal donde se ejecuta el aplicativo, de la forma: *[numero de linea]*: **Error**. Además se mostrarán del mismo modo los errores hallados durante el *Análisis Léxico*.

Ejemplo de Utilización

A continuación se adjuntan ejemplos que pueden ser utilizados para probar el aplicativo junto con la salida del analizador.

```
1 prueba;{ Falta la palabra reservada
      PROGRAM }
2 var
3   a,b:integer;
4 begin
5    a:=9;
6 end.
```

Fig. 3.4: Ejemplo de código sin palabra reservada.

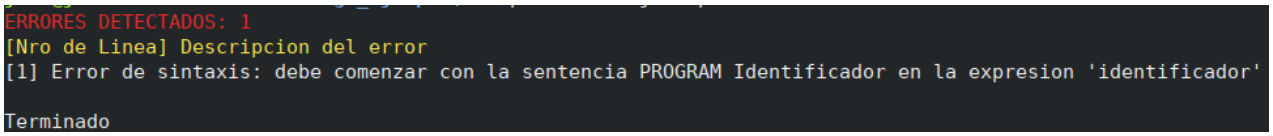

Fig. 3.5: Salida del ejemplo 3.4 con errores

```
1 program ; {Falta el IDENTIFICADOR de
      PROGRAM}
2 var a,b:integer;
3 begin
4    a:=9;
5 end.
```

Fig. 3.6: Ejemplo de código sin nombre del programa.

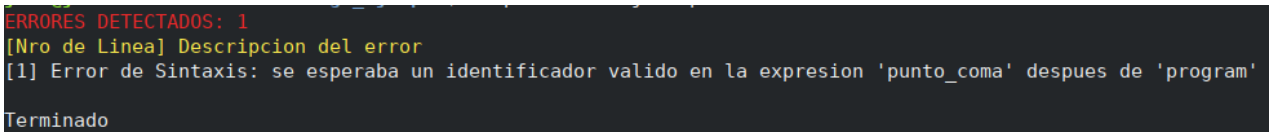

Fig. 3.7: Salida del ejemplo 3.6 con errores

```
1 program prueba; {Falta la palabra
      reservada VAR}
2   a,b:integer;
3 begin
4    a:=9;
5 end.
```

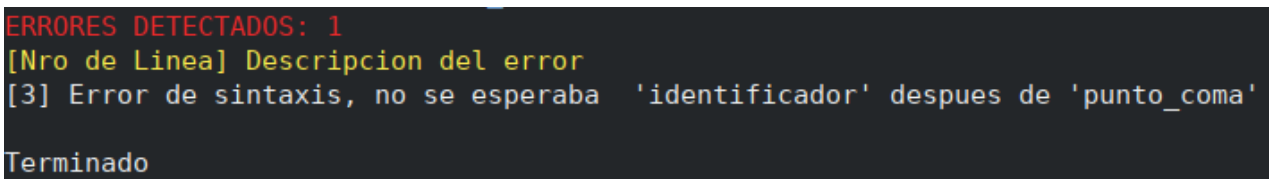

Fig. 3.8: Salida del ejemplo con error

```
1 program prueba; {Faltan los dos puntos
      antes de INTEGER}
2 var a, b integer;
3  begin
4    a:=9;
5    write( a );
6  end.
```

```
ERRORES DETECTADOS: 1
[Nro de Linea] Descripcion del error
[2] Error de sintaxis, no se esperaba  'integer' despues de 'identificador'

Terminado
```

Fig. 3.9: Salida del ejemplo con error

```
1 program prueba ;{ Palabra reservada VAR
      como IDENTIFICADOR }
2 var
3   a,b:integer;
4 function algo(x, var: integer): integer;
5  begin
6    algo := 10 + y - 10;
7  end;
8 begin
9   a:=9;
10   algo( a, 10 );
11 end.
```

```
ERRORES DETECTADOS: 1
[Nro de Linea] Descripcion del error
[4] Error de Sintaxis: se esperaba un identificador valido en la expresion 'var' despues de 'coma'

Terminado
```

Fig. 3.10: Salida del ejemplo con error

CONCLUSIONES

A lo largo del capítulo se ha descripto el funcionamiento genérico y el diseño e implementación propios del aplicativo de un *analizador sintáctico*.

Ha sido posible generar una implementación del funcionamiento de la gramática detallada en el capítulo 2 en lenguaje de programación *python* que se incorpora al aplicativo. Este *analizador sintáctico* realiza las derivaciones necesarias para lograr *árboles sintácticos implícitos* para el subconjunto del lenguaje *Pascal*. Éstos árboles, determinan las sentencias legales aceptadas y procesadas por el compilador y permiten generar errores y reportes precisos de las fallas que se pueden encontrar en el código fuente.

De esta forma se traduce el contenido *léxico-sintáctico* del código en árboles que alimentan el siguiente paso en el trabajo de un compilador: *el análisis semántico*, subiendo un paso más en el nivel de abstracción para poder generar, en un futuro, una traducción correcta en lenguaje máquina.

Capítulo 4
Analizador Semántico

En esta sección se detalla todo lo relevante en la definición e implementación del Analizador Semántico para el subconjunto reducido del lenguaje Pascal. Esto incluye una breve descripción de los tipos de chequeos que realiza el analizador semántico, el diseño de la Tabla de Símbolos, el diseño y la aplicación de las reglas semánticas asociadas a la gramática ya definida y ejemplos de ejecución del mismo.

Descripción del Problema

A partir del Analizador Sintáctico, especifique una estructura de datos para implementar la Tabla de Símbolos. Puntos a tener en cuenta:

(a) La tabla debe contener toda la información necesaria (acerca de los nombres que aparecen en el programa fuente) para implementar un control semántico completo.

(b) Debe permitir la representación de la información de ámbito de los subprogramas, teniendo en cuenta las reglas de alcance del subconjunto del lenguaje Pascal.

(c) La búsqueda y recuperación de información debe ser eficiente.

Incorporar, al Analizador Sintáctico desarrollado, las acciones requeridas para llevar a cabo la verificación semántica sobre el lenguaje especificado. El producto final (aplicativo) solicitado debe realizar los chequeos estáticos necesarios.

Descripción General

El Analizador Semántico trabaja directamente sobre el árbol sintáctico (implícito) generado por el analizador léxico (y la máquina de estados previa) y, junto con la información de la Tabla de Símbolos (TS), verifica la consistencia semántica entre el programa fuente y la definición del lenguaje. Los chequeos de consistencia semántica se basan principalmente en el chequeo de tipos, chequeo de unicidad y comprobación de parámetros cuando se requiera.

El compilador debe verificar que los tipos de datos de los operandos se correspondan con el tipo de la operación a ejecutar.

Podemos resumir las responsabilidades del analizador semántico en las siguientes tareas:

- Chequeo de flujo de control.

- Chequeo de unicidad.
- Correspondencia de nombres en más de un lugar. Por ejemplo, identificadores no pueden tener dos definiciones distintas y deben ser declarados antes de ser usados en alguna sentencia.
- Correspondencia entre argumentos formales y actuales en definiciones y llamadas a funciones y procedimientos.
- Recolección de información adicional para la Tabla de Símbolos.
- Generación de reportes al usuario sobre errores encontrados.

Es necesario aclarar que el analizador semántico también realiza algunos otros chequeos dinámicos, como por ejemplo límites de arreglos y punteros. Sin embargo, estos tipo de chequeos están por fuera del alcance de esta implementación.

ESTRATEGIAS UTILIZADAS PARA LA RESOLUCIÓN DEL PROBLEMA

Como se ha mencionado anteriormente la sintaxis de un lenguaje de programación describe el formato apropiado de sus programas, mientras que la semántica del lenguaje define lo que sus programas significan, es decir, lo que hace cada programa cuando se ejecuta.

Esto lleva a deducir que la especificación de la semántica de un lenguaje es más compleja de describir que la sintaxis del mismo. Para ello se utilizarán descripciones informales y ejemplos sugerentes.

Por este motivo en la sección se describe la generación de reglas semánticas, a través de un árbol de derivación y sus anotaciones.

<*mientras*> →	**while** <*expresionGeneral*> **do** <$mientras_1$>	{ **if** <*expresionGeneral*>.type = boolean **then** <*mientras*>.type:= $mientras_1$.type **else** <*mientras*>.type:= type_error }

Table 4.1: Reglas semánticas para la estructura *while*

<*mientras1*> →	**begin** <*expresionGeneral*> **end**	{ **if** <*expresionGeneral*>.type != error **then** <*mientras1*>.type:= *expresionGeneral*.type **else** <*mientras1*>.type:= type_error }

Table 4.2: Reglas semánticas para la estructura *while* con sentencias compuestas

<*mientras1*> →	<*expresionGeneral*>	{ **if** <*expresionGeneral*>.type != error **then** <*mientras1*>.type:= *expresionGeneral*.type **else** <*mientras1*>.type:= type_error }

Table 4.3: Reglas semánticas para la estructura *while* con una sentencia simple

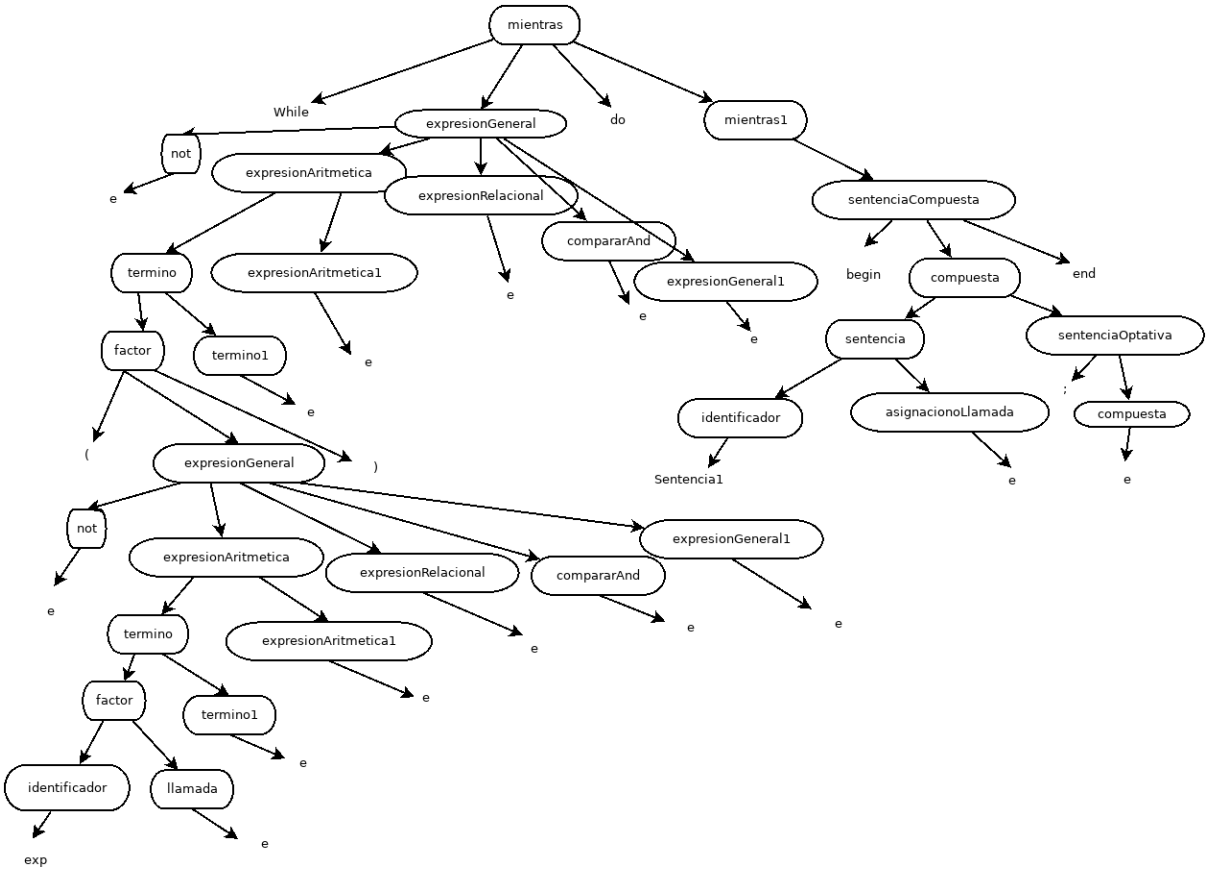

Fig. 4.1: Árbol de derivación de la estructura *while* con expresión compuesta

<*exp.General*> →	<*not*> <*exp.Aritmetica*> <*exp.Relacional*> <*comp.And*> <*exp.General1*>	{<*exp.Relacional*>.inh = <*exp.Aritmetica*>.type | <*compararAnd*>.inh = <*exp.Relacional*>.type | <*exp.General1*>.inh = <*comp.And*>.type | **if** <*not*>.type = boolean **and** <*exp.General1*>.type = boolean **then** <*exp.General*>.type:= boolean **else if** <*not*>.type = boolean **and** <*exp.General1*>.type != boolean **then** <*exp.General*>.type:= type_error **else** <*exp.General*>.type:= <*exp.General1*>.type }

Table 4.4: Reglas semánticas para el no-terminal *expresionGeneral*

<*expresionAritmetica*> →	<*termino*> <*expresionAritmetica1*>	{ <*expresionAritmetica1*>.inh := <*termino*>.type | <*expresionAritmetica*>.type := <*expresionAritmetica1*>.type }

Table 4.5: Reglas semánticas para el no-terminal *expresionAritmetica*

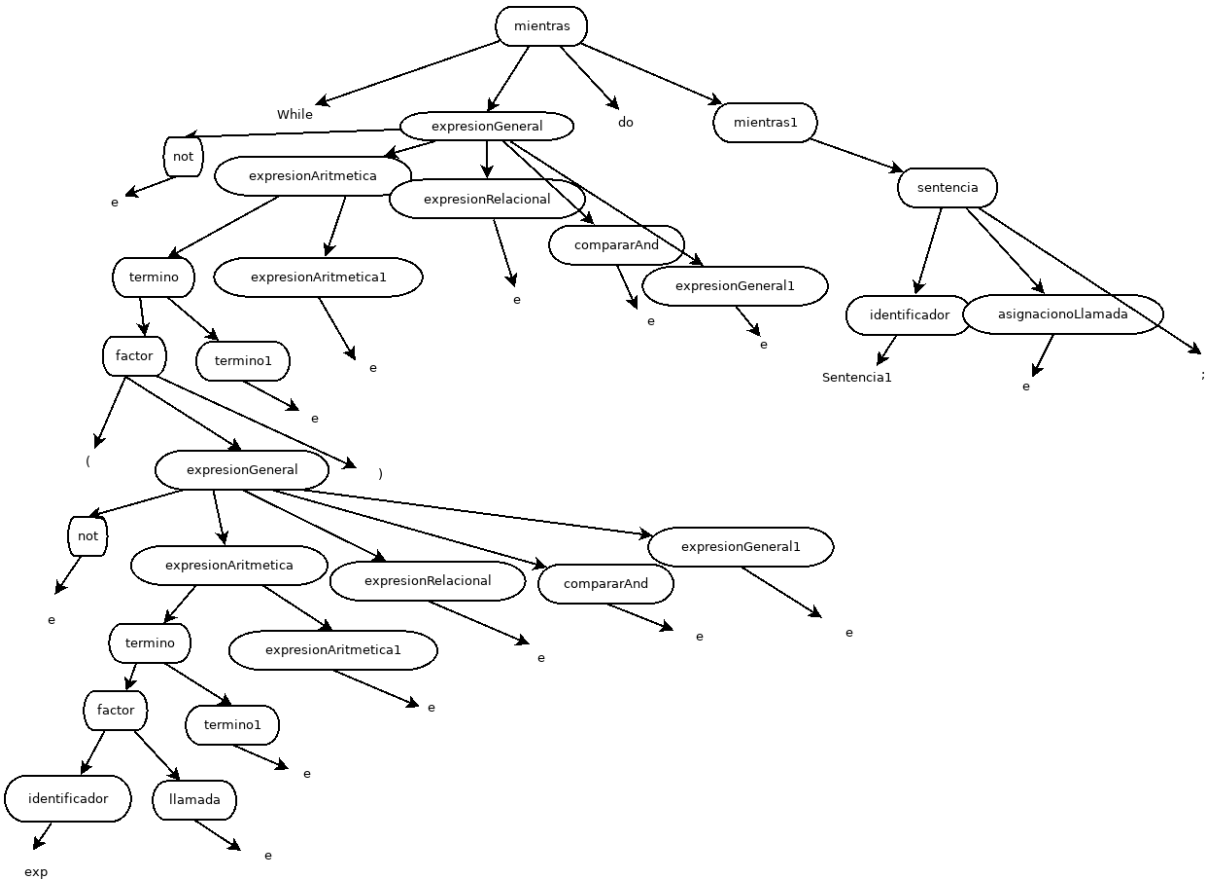

Fig. 4.2: Árbol de derivación de la estructura *while* con expresión simple

<*termino*> →	<*factor*>	{ <*termino1*>.inh := <*factor*>.type
	<*termino1*>	**|** <*termino*>.type := <*termino1*>.type }

Table 4.6: Reglas semánticas para el no-terminal *termino*

<*factor*> →	<*identificador*>	{ **if** <*identificador*>.type != type_error **then**
	<*llamada*>	<*factor*>.type = <*llamada*>.type
		else <*factor*>.type := type_error }

Table 4.7: Reglas semánticas para el no-terminal *factor*

ifthen →	**if** *expresionGeneral*	{ **if** *expresionGeneral*.type = boolean
	then *ifthen1*	**then** *ifthen*.type:= *ifthen1*.type
		else *ifthen*.type:= type_error }

Table 4.8: Reglas semánticas para la estructura *if*

PROBLEMAS ENCONTRADOS

Se tomaron diversas consideraciones sobre casos particulares que presentaron dificultades. Estos casos se detallan a continuación.

Dentro del mismo ambiente no pueden existir identificadores repetidos, tal como establece el chequeo de tipos. Teniendo en cuenta que cada uno de los identificadores puede tomar los siguientes tipos (no necesariamente el mismo): variable, nombre de procedimiento, nombre de función o del nombre del ambiente. Esto requiere que se busque por *nombre* en la tabla de símbolos con el fin de verificar la existencia de alguna definición previa coincidente con tal identificador.

Dentro de la lista de parámetros actuales/formales de un procedimiento o función no se pueden repetir los identificadores, pero cualquiera de ellos si puede ser igual a algún identificador definido en el ambiente que "contiene" al procedimiento o función -sobrecarga por re-definición de nombre-. Para este caso particular es necesario incorporar todos los parámetros cómo variables dentro del ámbiente del subprograma e igualmente identificarlos en el ambiente del *padre* del subprograma como parámetros asociados al nombre del procedimiento o función.

Asímismo, el llamado de naturaleza *recursiva* de un subprograma, fue motivo de dificultades en lo referente al chequeo de nombres.

Finalmente, fue necesario definir como *atributo* la variable de retorno asociada a las *funciones* para poder restringir que únicamente puede aparecer bajo el tipo *variable* del lado izquierdo de una asignación, limitando la aparición del nombre de la función a su utilización como subprograma del lado derecho de cualquier sentencia.

TABLA DE SÍMBOLOS

La Tabla de Símbolos (*TS*) es una estructura de datos empleada por el compilador para mantener información referida a nombres que aparecen en el programa fuente.

La información es recuperada de manera incremental durante todas las etapas de análisis, y usada por las fases de esta etapa (análisis léxico, sintáctico, semántico y generación de código intermedio) y por las fases de la etapa de síntesis (optimización y generación de código) para la posterior generación de código.

Normalmente, las entradas en la *TS* son creadas a medida que se procesan las declaraciones para los nombres -de variables, procedimientos y funciones-.

Las entradas en la *TS* contienen información relevante para el uso del nombre, ésta información en general es:

- Lexema.
- Atributo: si es una palabra reservada, una variable, un tipo, un procedimiento o una constante.
- Tipo de dato.
- Información de ambiente o alcance.
- Posición relativa, tamaño requerido en memoria.
- Información específica dependiente del atributo, cómo puede ser cantidad de parámetros, tipo de variable, una variable de retorno de una función.
- Otra información relevante.

Diseño

Ante la variedad y heterogeneidad de las información que almacena cada entrada de la *TS* es indispensable que la estructura que al implementa sea flexible como para almacenar toda

ésta información. La Tabla de Símbolos debería diseñarse como un Tipo de Dato Abstracto (*TDA*) que constituye una estructura global utilizada por distintos módulos del compilador.

Además de toda la información antes mencionada, las *TS* debe soportar las siguientes operaciones principales:

- Insertar información.
- Buscar o Consultar.
- Modificar o Actualizar valores de atributos.
- Eliminar información.

Para implementar la *TS* se requiere una estructura que permita llevar a cabo las operaciones mencionadas de manera eficiente.

Implementación

Como se menciono anteriormente, una *Tabla de Símbolos* requiere soportar datos heterogéneos de forma eficiente para su acceso y carga.

Ejemplos de estructuras de implementación de la Tabla de símbolos son: listas lineales, tablas hash, árboles entre otras estructuras. Para lenguajes como Pascal, que permiten tener subprogramas anidados, es habitual mantener una *TS* independiente por cada alcance o ámbito.

Para la implementación se utilizó una tabla **hash** (que corresponde con la clase *Dictionary* proporcionada por el lenguaje *Python*). Específicamente se creó una clase llamada Tabla, la cual puede ser utilizada por todas las clases que componen las distintas fases del compilador.

Se utiliza una *TS* individual para cada alcance, es decir, una nueva Tabla de Símbolos es creada cuando un ambiente nuevo en el programa es definido. El compilador guarda un puntero (de referencia) a la *TS* actual.

Las entradas en la *TS* contienen toda información que se considera relevante para el uso de los nombres que se van detectando, también almacenadas mediante diccionarios Hash de Python internos a la entrada de la *TS*. Para la realización del trabajo, se consideraron los siguientes campos:

- Nombre.
- Atributo (si es una variable, una función, un procedimiento, etc.).
- Tipo de dato (para la variable actual o valor retornado en caso de función).

- Lista de parámetros (lista de parámetros de procedimiento o función y tipo de cada uno de ellos).
- Nivel del ambiente.
- Referencia a la tabla de símbolos del programa, procedimiento o función padre (el que contiene al ambiente que genera ésta *TS*).

Luego, al momento de realizar la búsqueda de un nombre, la clase Tabla se encarga de dicha búsqueda.

El algoritmo de búsqueda comienza por acceder a la tabla de símbolos del ambiente actual (que se está analizando), por lo que se tiene acceso directo a ella. Si el nombre buscado es encontrado en la *TS* la búsqueda finaliza. En caso contrario, si el nombre no es encontrado, se continúa la búsqueda en la siguiente *TS* utilizando la referencia a la tabla de símbolos del programa, procedimiento o función padre de forma recursiva.

Esta búsqueda es eficiente al estar consultando sobre una Tabla Hash que funciona de forma análoga a un diccionario, permitiendo obtener en O(1) -Orden Constante- la respuesta a la búsqueda en cada ámbiente. Asímismo, se puede consultar en orden constante cualquier dato relevante de la entrada a través de su clave (o *key*).

Para una mejor comprensión de la implementación lograda, se toma como ejemplo la siguiente definición:

```
1 ⋮
2 function F1 (a:integer): Boolean;
3 begin
4     procedure P1(parametro1: integer, parametro2:boolean);
5         ⋮
6
7 var
8     b:boolean;
9 ⋮
10 end.
```

Fig. 4.3: Definición de una función de ejemplo F1

Si se analiza la *Tabla de Símbolos* de la definición anterior de forma gráfica, se podría observar la siguiente información almacenada en un diccionario.

Tabla de la Función F1			
Nombre	**Atributo**	**Tipo**	**Parametros**
P1	procedure	*Boolean*	{parametro1: Integer, parametro2: Boolean}
A	parametro	*Integer*	
B		*Boolean*	
F1	Retorno	*Boolean*	
Puntero de Retorno a TS superior			

Table 4.9: Ejemplo de la *Tabla de Símbolos* para el ejemplo 4.3

Es necesario aclarar que la "columna" *parámetros* contiene otro diccionario para acceder eficientemente a la información de cada una de las variables.

APLICATIVO

En esta sección se refiere a todo lo relativo al producto final en cuestión, desde su utilización y especificación, así como casos de usos, preguntas frecuentes y documentación apropiada del mismo.

Como ya detalla el capítulo anterior, el criterio del aplicativo es educacional e incremental, por lo que toda especificación ya detallada, es conservada y aplicada.

Por la naturaleza propia del analizador semántico, su ejecución requiere del analizador sintáctico y léxico implementado anteriormente, por lo que se podrá observar en su ejecución detalles de lo descripto en la utilización del analizador sintáctico.

Cabe destacar que los analizadores sintáctico y semántico se encuentran en el mismo archivo y conservan la comunicación con el analizador léxico detallada en capítulos anteriores.

A continuación se detalla el *Manual de Uso* del Aplicativo, la *Especificación de tratamiento de errores* en un nivel muy simple y finalmente *Casos de Uso* y ejemplos para ejecutar el analizador semántico.

Manual de Uso

El analizador fue desarrollado íntegramente en lenguaje de programación *Python*, por lo que se requiere que el sistema operativo objetivo tenga instalado *Python 2.7*.[1] En el caso de que

[1] Los sistemas GNU/Linux en su mayoría incorporan por defecto en su instalación básica estas librerías

este corriendo un sistema operativo que no tenga *Python* instalado, referimos al siguiente enlace de instalación oficial `https://www.python.org/downloads/`

Para su utilización, se distribuye el aplicativo en ejecutables *multi-arquitectura* para sistemas operativos GNU/Linux.

Para la ejecución del Analizador Semántico basta con ingresar en la carpeta donde se encuentre el aplicativo y ejecutarlo por Terminal de la siguiente forma:

```
1 $ ./aplicativo archivoDePrueba.ext
2
```

Fig. 4.4: $ simboliza el puntero del interprete de la terminal y archivoDePrueba.ext es el fichero a analizar

El resultado de la ejecución del aplicativo puede observarse siempre por salida estándar de la *Terminal* o *Consola*.

De forma análoga a lo detallado en el capítulo anterior, existen otras posibilidades de ejecución, mediante *flags* o parametros opcionales que permiten la ejecución *verbosa* del aplicativo. Ésta incluye todas las impresiones de *debugging* y salidas de control diferenciadas según el nivel de detalle requerido. Para ello basta con añadir el parámetro **-v** como muestra la figura 5.2

```
1 $ ./aplicativo -v archivoDePrueba.ext
2
```

Fig. 4.5: $ simboliza el puntero del interprete de la terminal, -v indica *ejecución verbosa de grado 1* y archivoDePrueba.ext es el fichero a analizar

El aplicativo cuenta con dos niveles de detalle para la salida *verbosa*. El primer nivel, donde únicamente se indica una bandera (‘*-v*’) para la impresión que detalla los mensajes del analizador Semántico en conjunto con el analizador Sintáctico y un segundo nivel de detalle, que se identifica con dos banderas (‘-vv’) donde además de lo detallado por el primer nivel, se añaden las impresiones del *Analizador Léxico*.

Dentro de las impresiones generadas por la ejecución verbosa, es posible identificas las referidas al analizador semántico en color verde e identificadas por una etiqueta de comienzo de linea: *"[Semántico]"*.

Es posible que requiera permisos de ejecución para poder utilizar el aplicativo. Si falla su ejecución por estas causas, para solucionarlo basta con ejecutar los siguientes comandos:

```
$ chmod +x aplicativo
```

Fig. 4.6: Brindar permisos de ejecución al aplicativo por Terminal

Y luego continuar con los pasos descriptos anteriormente.

Errores

Los errores semánticos detectables por el aplicativo desarrollado están enfocados en los alcanzados por el análisis estático a través del *chequeo de tipos* y *chequeo de unicidad*.

Para esto, cada vez que se encuentra una definición de nombres, se accede a la *Tabla de Símbolos* para verificar si el mismo ya fue definido anteriormente en algún ambiente alcanzable por cadena estática dentro del entorno de la definición.

De forma análoga, cada vez que aparece un nombre en el análisis del archivo fuente, se determina el *tipo* del mismo sobre la entrada pertinente en la *TS*.

Esto simplifica el análisis semántico y permite una detección y descripción de errores expresiva y significativa para el usuario.

Todos los errores encontrados en el análisis se muestran por salida estándar en la terminal donde se ejecuta el aplicativo, de la forma: *[numero de linea]*: **Error**. Además se mostrarán del mismo modo los errores hallados durante las etapas anteriores de análisis.

Ejemplo de Utilización

A continuación se adjuntan ejemplos de ficheros que pueden ser utilizados para probar el aplicativo junto con la resolución del analizador -si existen errores -.

```
1 PROGRAM prueba; {Con errores semanticos}
2 var
3 {Comentario seguramente util y perspicaz}
4 a,b : integer;
5 c : boolean;
6 d : boolean;
7 procedure funcion;
8 var
9 a : integer;
10 begin
11         a := 10<9; {Tipo incorrecto:
       Integer := Boolean}
12 end;
13 function p1(zz:integer):boolean;
14 var
15 a : integer;
16 begin
17 end;
18 procedure pa1(yy:boolean;xx:integer);
19 var
20 i,ii : integer;
21 begin
22 write(i+ii);
23 end;
24 procedure pa2;
25 var
26 i,ii:boolean;
27 begin
28 i:=i+ii; {Tipo incorrecto: Boolean :=
       Boolean + Boolean}
29 end;
30 function pas:boolean;
31 var
32 t:integer;
33 t,parametro3,v:boolean; {Identificador ya
       definido}
34 begin
35         a := 10+9;
36         v := pas or true;
37 end;
38 BEGIN
39 pa2(a,b); {Cantidad de Parametros
       incorrectos: Sobran parametros}
40 if not (a > b) or pas THEN
41         c := true or false
42 else
43         e:=false; {Identificador no
       declarado}
44 noidentado; {Identificador no declarado}
45 a := a * b < true; {Error de Tipo:
       Integer < Boolean}
46 pa1; {Cantidad de parametros incorrectos:
       Faltan parametros}
47 END.
```

Fig. 4.7: Ejemplo de programa con errores semánticos

```
ERRORES DETECTADOS: 8
[Nro de Linea] Descripcion del error
[12] La asignación no es correcta. El identificador 'a' no es del tipo correcto.

[32] El operador + se encuentra definido para tipos INTEGER.

[38] Identificador 't' ya fue definido anteriormente

[46] Error: Cantidad de parametros incorrecto. No se esperaban parametros

[50] El identificador 'e' no esta definido.

[51] El identificador 'noidentado' no esta definido.

[52] Tipo Incompatible: el operador < se encuentra definido para tipo INTEGER.

[53] Error: Cantidad de parametros incorrecto. Se esperaba(n) 2 parametro(s)

Terminado
```

Fig. 4.8: Salida del ejemplo 4.7 con múltiples errores

CONCLUSIONES

Analizando toda la información descripta en el capítulo anterior, se puede afirmar que no sólo es factible una implementación de un analizador semántico para el subconjunto de lenguaje *Pascal*, sino que ha sido lograda a partir de lo implementado desde el analizador sintáctico (ver capitulo 3).

Se logró un **analizador semántico** capaz de trabajar con árboles sintácticos implícitos, añadiendo reglas semánticas precisas y atributos heredados y sintetizados que fueron implementados a nivel código. En conjunto con la implementación de la *Tabla de Símbolos* el resultado final fue exitoso y logra la verificación de la semántica para un código fuente dado.

Esto es esencial para el siguiente paso propio de un compilador y da inicio a la *generación de código intermedio*.

Capítulo 5

Generador de Código Intermedio

En esta sección se detalla todo lo relevante en la definición e implementación del Generador de Código Intermedio, cuyo lenguaje origen es el subconjunto reducido del lenguaje Pascal y lenguaje destino es *MEPA*. Esto incluye una breve descripción del proceso de traducción, tipos de sentencias, código de tres direcciones, manejos de alcances y ejemplos de ejecución del mismo. Finalizando con una conclusión sobre el trabajo realizado y las limitaciones que presenta esta implementación.

Descripción del Problema

A partir del Analizador Semántico desarrollado, extienda el mismo para que implemente un Generador de Código Intermedio tomando como destino el lenguaje MEPA que cumpla con los requisitos de portabilidad ya detallados en las secciones anteriores.

Descripción General

Luego del análisis sintáctico y semántico, el compilador debe generar un código de bajo nivel. Si se considerase una máquina abstracta como lenguaje destino, podemos generar código de bajo nivel fácil de producir y traducir en el lenguaje de la máquina destino final. Este tipo de código es denominado *código intermedio*.

El código intermedio es independiente de la máquina destino, pero conceptualmente, provee un nivel de abstracción más cercano a ella, sirviendo de facilitador para una posterior traducción al código específico de la máquina a utilizar. Esta etapa realiza la traducción del programa en lenguaje fuente a un programa equivalente en código intermedio .

La generación de un código intermedio tiene varios beneficios:

- El análisis se independiza del lenguaje destino.
- Es posible aplicar optimizaciones independientes del lenguaje destino.
- Portabilidad. Un compilador para una máquina destino diferente requiere modificar solo algunos componentes.

El aplicativo realizado traduce el programa fuente (lenguaje de programación Pascal) en un código intermedio para una máquina hipotética denominada sistema de ejecución para el lenguaje Pascal. Esta máquina es la *M.E.Pa* (Máquina de Ejecución para Pascal).

La principal complejidad esperada en esta etapa es el trato de expresiones aritméticas y booleanas. Dicha complejidad viene de la asociatividad, la precedencia de operadores y el uso de paréntesis.

Como ejemplo, consideramos la siguiente expresión: $10 + 2 * 5$. Aquí, primero se realiza la multiplicación entre 2 y 5, y finalmente este resultado es sumado a 10. El código *M.E.Pa* para esta expresión es el siguiente:

```
APCT 5
APCT 2
MULT
APCT 10
SUMA
```

Desde este código y considerando la característica de máquina de pila de la *MEPA*, podemos inferir que la expresión posfija asociada a $10 + 2 * 5$ nos proporciona una forma de tratar esta complejidad en expresiones. La notación posfija para $10 + 2 * 5$ es: $10\ 2\ 5\ *+$.

En la siguiente sección se detalla la estrategia utilizada para lograr la traducción a *M.E.Pa* con la notación antes mencionada.

ESTRATEGIAS UTILIZADAS PARA LA RESOLUCIÓN DEL PROBLEMA

La guía para implementar esta etapa fue el apunte proporcionado por la cátedra "Compiladores e Intérpretes e Máquina de Ejecución de Pascal – M.E.Pa."[2]. En el apunte se explica con detalle como es el manejo de la memoria por M.E.Pa., el repertorio de instrucciones, como usarlas, y ejemplos de uso.

La fase de generación de código intermedio implica trabajar con el repertorio de instrucciones de la *M.E.Pa* e ir generando el equivalente de los constructores del lenguaje fuente a medida que el programa a compilar está siendo procesado.

Por la naturaleza propia del procesamiento del lenguaje y la estructura de los analizadores sintáctico y semántico que transforman el código fuente en árboles de derivación. La generación del código *M.E.Pa* fue lograda fácilmente al ir acompañando el proceso de construcción del árbol semántico con la generación temporal de código que se va encolando a medida

que se procesa el código fuente, simplificando enormemente la tarea de traducción de las sentencias escritas en formato *infijo* al requerido formato *posfijo*.

Esto permitió aprovechar el movimiento propio del árbol para heredar valores de las etiquetas de salto e ir generando código en cada llamado interno del aplicativo.

Una vez resuelta la dificultad inicial, las particularidades de la implementación se pueden resumir en dos grandes tareas: la definición de los subprogramas y el cálculo de dirección de las variables. Estas tareas requirieron más tiempo que las demás sentencias debido al manejo de las etiquetas, las llamadas y retornos a los subprogramas.

Para mantener la coherencia en la numeración de las *etiquetas* (en particular en el anidamiento de *sentencias condicionales*) se optó por realizar pasaje de parámetros entre métodos internos del analizador para instanciar los valores necesarios.

Para lograr el funcionamiento correcto de los identificadores, fue necesario ampliar los atributos almacenados en la *Tabla de Símbolos*, adjuntando para los subprogramas los valores de *nivel* de cada subprograma.

De forma análoga, en cada variable se almacenó su *nivel* de acuerdo al procedimiento que lo define y su desplazamiento (llamado *dirección* en la implementación interna de la *TS*) relativo a la cantidad de variables definidas en cada entorno (ya que los tamaños de los tipos de variables *Boolean* e *Integer* es 1).

En lo que respecta a la generación de código, se utiliza un acceso global a una variable que permite ir '*encolando*' el código generado, para que al finalizar el correcto procesamiento del archivo fuente, se guarde todo lo generado en un archivo externo de extensión *'.mepa'*, priorizando la legibilidad del código.

LIMITACIONES DEL APLICATIVO

Las limitaciones del aplicativo radican en el subconjunto del lenguaje utilizado. El funcionamiento del aplicativo es preciso para el subconjunto acotado del lenguaje *Pascal*, es decir, se excluyen los tipos de datos definidos por el usuario, el manejo de arreglos, la impresión de múltiples variables en la misma sentencia, los tipos de datos se limitaron a *integer* y *boolean*.

Exceptuando todas las limitaciones detalladas, el funcionamiento es correcto.

APLICATIVO

En esta sección se detalla todo lo relativo al producto final en cuestión, desde su utilización y especificación, así como casos de usos, preguntas frecuentes y documentación apropiada del mismo.

Como ya detalla el capítulo anterior, el criterio del aplicativo es educacional e incremental, por lo que toda especificación ya detallada, es conservada y aplicada.

A continuación se detalla el *Manual de Uso* del Aplicativo, la *Especificación de tratamiento de errores* en un nivel muy simple y finalmente *Casos de Uso* y ejemplos para ejecutar el analizador semántico.

Manual de Uso

El analizador fue desarrollado íntegramente en lenguaje de programación *Python*, por lo que se requiere que el sistema operativo objetivo tenga instalado *Python 2.7*.[1] En el caso de que este corriendo un sistema operativo que no tenga *Python* instalado, referimos al siguiente enlace de instalación oficial `https://www.python.org/downloads/`

Para su utilización, se distribuye el aplicativo en ejecutables *multi-arquitectura* para sistemas operativos GNU/Linux.

Para la ejecución del Generador de Código intermedio basta con ingresar en la carpeta donde se encuentre el aplicativo ya descripto anteriormente que incluye todo el trabajo del compilador y ejecutarlo por Terminal de la siguiente forma:

```
1 $ ./aplicativo archivoDePrueba.ext
```

Fig. 5.1: $ simboliza el puntero del interprete de la terminal y archivoDePrueba.ext es el fichero a analizar

El resultado de la ejecución del aplicativo puede observarse siempre por salida estándar de la *Terminal* o *Consola*.

Asímismo, la ejecución del aplicativo ahora genera un archivo, cuyo nombre es el mismo que el archivo pasado por parámetro para analizar y la extensión es '*.mepa*' (en el ejemplo 5.1. el archivo generado será archivoDePrueba.ext.mepa). El mismo contiene la traducción del archivo ingresado lista para ser ejecutada en la máquina virtual.

[1] Los sistemas GNU/Linux en su mayoría incorporan por defecto en su instalación básica estas librerías

De forma análoga a lo detallado en el capítulo anterior, existen otras posibilidades de ejecución, mediante *flags* o parámetros opcionales que permiten la ejecución *verbosa* del aplicativo.

```
1 $ ./aplicativo -v archivoDePrueba.ext
```

Fig. 5.2: $ simboliza el puntero del interprete de la terminal, -v indica *ejecución verbosa de grado 1* y archivoDePrueba.ext es el fichero a analizar

Es posible que requiera permisos de ejecución para poder utilizar el aplicativo. Si falla su ejecución por estas causas, para solucionarlo basta con ejecutar los siguientes comandos:

```
1 $ chmod +x aplicativo
```

Fig. 5.3: Brindar permisos de ejecución al aplicativo por Terminal

Y luego continuar con los pasos descriptos anteriormente.

Ejemplo de Utilización

En la siguiente sección se adjuntan ejemplos de ficheros que pueden ser utilizados para probar el aplicativo junto con la resolución del analizador / generador de código intermedio.

Variables locales de un subprograma

```
1 PROGRAM prueba;
2 var
3 {Comentario seguramente util y perspicaz}
4 a,b : integer;
5 c : boolean;
6 d : boolean;
7
8 function p1(zz:integer):boolean;
9 var
10 a : integer;
11
12 procedure pa1(yy:boolean;xx:integer);
13 var
14 i,ii : integer;
15 begin
16 xx := i + ii;
17 write(i+ii);
18 end;
19
20 procedure pa2;
21 var
22 i,ii:boolean;
23 begin
24 i:=i or ii;
25 end;
26
27 function pa3(ww:integer;www:integer;u:
       integer):boolean;
28 var
29 t:integer;
30 v:boolean;
31 begin
32 {t := u and v;}
33 end;
34
35 begin
36         a := 10-9;
37         p1 := true;
38 end;
39
40 BEGIN
41 while (a < 10) do
42 begin
43 c:=3+b>6 or (2<>a and a<4);
44 end;
45 d:=c and false;
46 if not(a > b) THEN
47         c := true or false
48 else
49         c:=false;
50 END.
```

Fig. 5.4: Ejemplo de ejecución con variables locales de subprogramas y con numerosas sentencias de operaciones simples

```
1  INPP
2  RMEM 4
3  DSVS L1
4  L2 ENPR 1
5  RMEM 1
6  L3 ENPR 2
7  RMEM 2
8  APVL 2,0
9  APVL 2,1
10 SUMA
11 ALVL 2,-3
12 APVL 2,0
13 APVL 2,1
14 SUMA
15 IMPR
16 LMEM 2
17 RTPR 2,2
18 L4 ENPR 2
19 RMEM 2
20 APVL 2,0
21 APVL 2,1
22 DISJ
23 ALVL 2,0
24 LMEM 2
25 RTPR 2,0
26 L5 ENPR 2
27 RMEM 2
28 LMEM 2
29 RTPR 2 3
30 APCT 10
31 APCT 9
32 SUST
33 ALVL 1,0
34 APCT 1
35 ALVL 1,-4
36 LMEM 1
37 RTPR 1 1
38 L1 NADA
39 L6 NADA
40 APVL 0,0
41 APCT 10
42 CMME
43 DSVF L7
44 APCT 3
45 APVL 0,1
46 SUMA
47 APCT 6
48 CMMA
49 APCT 2
50 APVL 0,0
51 CMDG
52 APVL 0,0
53 APCT 4
54 CMME
55 CONJ
56 DISJ
57 ALVL 0,2
58 DSVS L6
59 L7 NADA
60 APVL 0,2
61 APCT 0
62 CONJ
63 ALVL 0,3
64 APVL 0,0
65 APVL 0,1
66 CMMA
67 NEGA
68 DSVF L8
69 APCT 1
70 APCT 0
71 DISJ
72 ALVL 0,2
73 DSVS L9
74 L8 NADA
75 APCT 0
76 ALVL 0,2
77 L9 NADA
78 LMEM 4
79 PARA
```

Fig. 5.5: Código MEPA producido por el aplicativo para el ejemplo 5.4

Pasaje de parámetros para una función

```
1  Program ejemplo;
2  var m: integer;
3  function f(n:integer; k:integer) :
       integer;
4  var p,q:integer;
5  begin
6  if n<2 then
7      begin
8      f:=n;
9      p:=0
10     end
11 else
12     begin
13     f:= f(n-1,p)+f(n-2,q);
14     p:= p+q+1
15     end;
16 write(n);
17 write(k)
18 end;
19
20 begin
21     write(f(3,m));
22     write(m)
23 end.
```

Fig. 5.6: Ejemplo con pasaje de parámetros en una función con variables locales y sentencia condicional

```
1   INPP
2   RMEM 1
3   DSVS L1
4   L2 ENPR 1
5   RMEM 2
6   APVL 1,-4
7   APCT 2
8   CMME
9   DSVF L3
10  APVL 1,-4
11  ALVL 1,-5
12  APCT 0
13  ALVL 1,0
14  DSVS L4
15  L3 NADA
16  RMEM 1
17  APVL 1,-4
18  APCT 1
19  SUST
20  APVL 1,0
21  LLPR L2
22  RMEM 1
23  APVL 1,-4
24  APCT 2
25  SUST
26  APVL 1,1
27  LLPR L2
28  SUMA
29  ALVL 1,-5
30  APVL 1,0
31  APVL 1,1
32  SUMA
33  APCT 1
34  SUMA
35  ALVL 1,0
36  L4 NADA
37  APVL 1,-4
38  IMPR
39  APVL 1,-3
40  IMPR
41  LMEM 2
42  RTPR 1 2
43  L1 NADA
44  RMEM 1
45  APCT 3
46  APVL 0,0
47  LLPR L2
48  IMPR
49  APVL 0,0
50  IMPR
51  LMEM 1
52  PARA
```

Fig. 5.7: Código MEPA producido por el aplicativo para el ejemplo 5.6

Sentencia Condicional

```
1  program Ejemplo;
2  var a,b, n, s : integer;
3  p,q :boolean;
4  begin
5      s:=0;
6      n:=9;
7      a:=10;
8      b:=20;
9      p:= true;
10     q:= true;
11
12     if a>b then
13         q := p and true
14     else
15         if (a < 10) then
16             q := true
17
18         else
19             q := false;
20     write(q);
21 end.
```

Fig. 5.8: Ejemplo de ejecución con una sentencia condicional

```
1   INPP          12  ALVL 0,1      23  CONJ          34  L4 NADA
2   RMEM 6        13  APCT 1        24  ALVL 0,5      35  APCT 0
3   DSVS L1       14  ALVL 0,4      25  DSVS L3       36  ALVL 0,5
4   L1 NADA       15  APCT 1        26  L2 NADA       37  L5 NADA
5   APCT 0        16  ALVL 0,5      27  APVL 0,0      38  L3 NADA
6   ALVL 0,3      17  APVL 0,0      28  APCT 10       39  APVL 0,5
7   APCT 9        18  APVL 0,1      29  CMME          40  IMPR
8   ALVL 0,2      19  CMMA          30  DSVF L4       41  LMEM 6
9   APCT 10       20  DSVF L2       31  APCT 1        42  PARA
10  ALVL 0,0      21  APVL 0,4      32  ALVL 0,5
11  APCT 20       22  APCT 1        33  DSVS L5
```

Fig. 5.9: Código MEPA producido por el aplicativo para el ejemplo 5.8

Sentencia Repetitiva

```
1 program Ejemplo;          7     n:=9;          13    while s<=n do
2 var a,b, n, s : integer;  8     a:=10;         14    begin
3 p,q :boolean;             9     b:=20;         15        s := s+3;
4                           10    p:= true;      16        write(s)
5 begin                     11    q:= true;      17    end
6     s:=0;                 12                   18 end.
```

Fig. 5.10: Ejemplo de ejecución con una sentencia repetitiva condicional

```
1   INPP          12  ALVL 0,1      23  APCT 3
2   RMEM 6        13  APCT 1        24  SUMA
3   DSVS L1       14  ALVL 0,4      25  ALVL 0,3
4   L1 NADA       15  APCT 1        26  APVL 0,3
5   APCT 0        16  ALVL 0,5      27  IMPR
6   ALVL 0,3      17  L2 NADA       28  DSVS L2
7   APCT 9        18  APVL 0,3      29  L3 NADA
8   ALVL 0,2      19  APVL 0,2      30  LMEM 6
9   APCT 10       20  CMNI          31  PARA
10  ALVL 0,0      21  DSVF L3
11  APCT 20       22  APVL 0,3
```

Fig. 5.11: Código MEPA producido por el aplicativo para el ejemplo 5.10

CONCLUSIONES

Luego de analizar y estudiar cuidadosamente la especificación de *M.E.Pa* y analizando las ventajas que se obtienen se pudo lograr una traducción desde el lenguaje de programación *Pascal*, añadiendo funcionalidad al aplicativo del compilador logrado hasta el momento.

Se ha podido efectuar la implementación de forma exitosa obteniendo un *Generador de Código Intermedio* dentro del mismo programa que hasta el momento contenía la funcionalidad de Analizador léxico, sintáctico y semántico completando todas las tareas propias del *backend* de un compilador dando por finalizada la etapa de análisis.

Capítulo 6

Conclusiones

Con esto finaliza la práctica profesional supervisada en el área de *Compiladores e Interpretes* para los autores del informe.

Esta práctica forma parte de los programas de estudio de las materias *Diseño de Compiladores e Interpretes* y *Laboratorio de Compiladores e Interpretes* de la *Licenciatura en Ciencias de la Computación*, en la Facultad de Informática de la *Universidad Nacional del Comahue*.

Se ha logrado el objetivo esperado, la especificación, el diseño y desarrollo de un compilador de una pasada para un subconjunto finito del lenguaje de programación *Pascal*. Se ha realizado un trabajo teórico de diseño, que se ha materializado en un producto de software con finalidad educativa capaz de realizar las tareas de **análisis léxico**, **análisis sintáctico**, **análisis semántico** y **generación de código intermedio** de forma efectiva y precisa.

Además, este trabajo sirve de documentación y guía de desarrollo para comprender y utilizar el producto final.

Todo el código del *aplicativo*, en conjunto con una copia de este informe, los gráficos incorporados al mismo y la licencia final del producto se encuentra disponible en: `https://github.com/jmdelafuente22/cei`.

Finalmente, es deseo de los autores que el trabajo realizado sirva como material de consulta para cualquier estudiante que inicie un trabajo similar, facilitando y dimensionando la labor que representa el desarrollo de un compilador así como ha sido de desarrollo personal e intelectual para los mismos.

Bibliografía

[1] AHO. *Compiladores: Principios, Tecnicas y Herramientas, 2 ed.* Pearson Education, 2008.

[2] AMAOLO, M. Compiladores e intérpretes máquina de ejecución de pascal - m.e.pa., 2015. Universidad Nacional del Comahue. Facultad de Informática.

[3] LEWIS, H. R., AND PAPADIMITRIOU, C. H. *Elements of the Theory of Computation*, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[4] LLAVORI, R., AND QUEREDA, J. *Introducción a la programación con Pascal.* Treballs d'informàtica i tecnologia. Universitat Jaume I, 2000.

[5] PARRA, G. Lenguajes libres de contexto: Propiedades, 2015. Universidad Nacional del Comahue. Facultad de Informática.

[6] PARRA, G. Lenguajes regulares: Propiedades algorítmicas y conclusiones, 2015. Universidad Nacional del Comahue. Facultad de Informática.

[7] WELSH, J., AND ELDER, J. *Introduction to PASCAL.* International series in computer science. Prentice/Hall International, 1979.