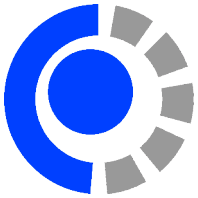
Universidad Nacional del Comahue
Facultad de Informática



Diseño de Compiladores e Intérpretes

Desarrollo de un Compilador para Pascal

GRUPO 1

Bermudez Martín

FAI-1140

Marinelli Giuliano

FAI-1196

{martin.bermudez, giuliano.marinelli}
@est.fi.uncoma.edu.ar

Neuquén Argentina

2018

Índice general

Capítulo 1

Introducción

1.1. Compiladores y proceso de desarrollo

Los compiladores son programas que permiten la traducción entre especificaciones de lenguajes de programación. Tienen el objetivo principal de traducir lenguajes de alto nivel en otros de mas bajo nivel que sean interpretables por la arquitectura del hardware en que se ejecutarán.

Para poder lograr esta tarea, los compiladores realizan varias etapas, donde, en un principio, se tiene como entrada el código del lenguaje a traducir el cuál pasará por las etapas que pueden verse en la figura 1.1 y finalmente retornará el código en el lenguaje objetivo.

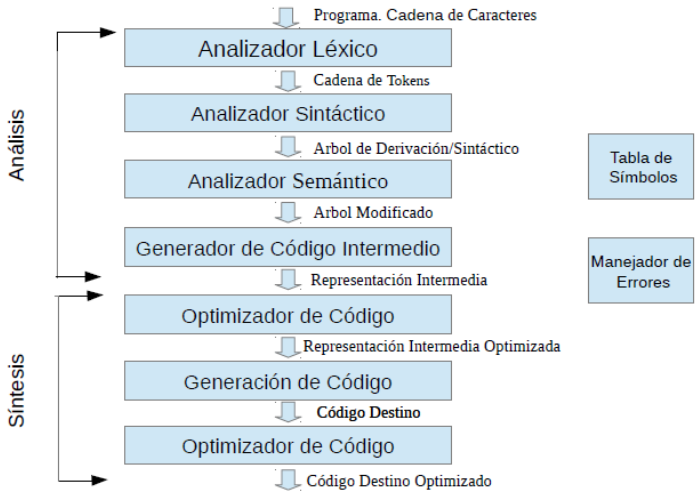


Figura 1.1: Etapas del desarrollo de un compilador, indicando los dos procesos principales.

El desarrollo tiene dos procesos principales, el *Análisis* y la *Síntesis*. El análisis es el proceso en que se dará énfasis, ya que consta de las etapas más importantes como el análisis léxico, sintáctico y semántico, teniendo principal hincapié en la sintaxis ya que se realizará una *traducción dirigida por la sintaxis* para este trabajo.

En este sentido, para este informe vamos a mostrar el diseño y la implementación de un compilador para un conjunto reducido de construcciones del lenguaje Pascal, explicando el desarrollo de cada etapa y cómo se relaciona con la etapa siguiente.

Capítulo 2

Gramática del lenguaje

2.1. Introducción

El diseño y la implementación de un compilador se compone de diferentes etapas, donde cada una requiere una entrada y produce una salida para la etapa siguiente. En este capítulo nos centraremos en la definición preliminar de una gramática para generar las cadenas de entrada que tendrá nuestro lenguaje. En capítulos posteriores utilizaremos estas definiciones para el desarrollo de las etapas siguientes en proceso de compilación.

2.2. Descripción del problema

Para esta primer etapa, debemos definir cómo va a ser la estructura de las cadenas de entrada del lenguaje. Para esto formalizaremos con una gramática utilizando una notación específica, que veremos en la sección 2.3. En este sentido, se definirán las siguientes construcciones que serán aceptadas por el compilador:

- Definición de variables y subprogramas.
- Tipos de datos simples: se utilizaron datos **integer** y **boolean**.
- Constantes usadas en las expresiones: **true** y **false**.
- Subprogramas: se pueden definir procedimientos y funciones, con pasaje de parámetro por valor.
- Sentencias:
  - Asignación.
  - Repetitiva: **while**.
  - Alternativa: **if then else**.
  - Sentencias compuestas: **begin end**.
- Procedimientos **read** y **write** para un solo valor.
- Operaciones aritméticas: $+$, $-$, $*$, $/$, con uso de paréntesis.
- Operaciones booleanas: operadores **AND**, **OR**, **NOT**, y operadores de comparación: $<=$, $>=$, $>$, $<$, $=$.

2.3. Metalenguaje

Para expresar el lenguaje se ha utilizado una gramática con la notación **BNF** sin extender, de manera que se eviten ambigüedades con respecto a las notaciones propuestas por los diferentes autores.

Hemos utilizado la siguiente convención para la definición de la gramática:

- Los símbolos no terminales van encerrados entre $<$ y $>$.

- Los símbolos terminales se ven en **negrita**.
- Cada regla de producción diferencia su parte izquierda de su parte derecha con el símbolo →.
- Las opciones alternativas en cada regla se especifican con el símbolo |.

2.4. Estrategias

Para definir la gramática hemos tenido en cuenta la propia definición dada para Pascal (que puede encontrarse en la web) y se la ha limitado en varios sentidos expresados en la sección 2.5, como también se ha optimizado para el caso de no terminales inútiles o repetidos, siempre buscando generar una gramática válida.

Por otro lado, como se ha decidido implementarla en **BNF** sin extender [2.3], la gramática quedó expresada utilizando recursividad, lo cual en etapas siguientes permitirá una traducción a código mas directa.

Se ha decidido realizar la gramática considerando optimizar el uso de los no terminales, en vez de tener en cuenta una posible expansión a futuro, es decir, que nuestra definición no será fácilmente adaptable al cambio pero tendrá un mejor rendimiento en el momento de implementarse.

2.5. Limitaciones

Como se ha mencionado en la sección 2.2, se realizará un compilador para el lenguaje Pascal limitado en varios aspectos. En un principio solo se tendrán en cuenta variables del tipo **integer** y **boolean**. Los procedimientos y funciones no podrán recibir parámetros por referencia, solo por valor. Dentro de las sentencias alternativas solo encontraremos **if then else** y en repetitivas solo la sentencia **while**. No se tendrá en cuenta parámetros enviados al programa principal. Y finalmente las sentencias de escritura y lectura se verán limitadas a un solo valor, y para la salida se tendrán en cuenta algunas consideraciones [2.5.1].

Con respecto a la especificación, debido a que hemos decidido utilizar **BNF** sin extender, este se ve limitado en expresividad, ya que no utilizamos símbolos para repetitivas, opcionales, agrupaciones, etc. A pesar de esto, creemos que la especificación es clara dado que el lenguaje trabajado no es de gran magnitud.

2.5.1. Consideraciones para la salida

La instrucción de salida write permite especificar el formato de salida de una variable. El formato depende del tipo de variable que se pase por parámetro. En nuestro caso tenemos solo dos tipos de variables, integer y boolean. Para imprimir estos dos tipos de dato hay que tener en cuenta las siguientes características:

- Integer
  write(number : fieldwidth);
  Se reservará un campo de ancho “fieldwidth” y “number” se justificará a la derecha en este campo, con los lugares restantes a la izquierda llenos de espacios en blanco. Si “fieldwidth” no es lo suficientemente grande como para cubrir todos los dígitos de “number”, se mostrará correctamente en un campo más amplio. “fieldwidth” también puede ser una variable o expresión de tipo entero.
- Boolean
  write(flag : fieldwidth);
  Esto generará “verdadero” o “falso” dentro del ancho de campo especificado, si se especifica.

El parámetro “fieldwidth” es opcional. Para facilitar la implementación no vamos a tener en cuenta esta característica del lenguaje.

2.6. Definición de la gramática

A continuación se mostrará la definición de la gramática en **BNF** para el lenguaje enunciado anteriormente. Considerar el símbolo inicial <program>.

<program> → <program-heading> <block> **.**

<program-heading> → **program** <identifier> **;**

<block> → <declaration-block> <multiple-statement> | <multiple-statement>

<declaration-block> → <variable-declaration-block> | <variable-declaration-block> <procedure-and-function-declaration-list> | <procedure-and-function-declaration-list>

<variable-declaration-block> → **var** <variable-declaration-list>

<variable-declaration-list> → <variable-declaration> **;** | <variable-declaration> **;** <variable-declaration-list>

<variable-declaration> → <identifier-list> **:** <type>

<procedure-and-function-declaration-list> → <procedure-declaration> **;** | <function-declaration> **;** | <procedure-declaration> **;** <procedure-and-function-declaration-list> | <function-declaration> **;** <procedure-and-function-declaration-list>

<procedure-declaration> → <procedure-heading> **;** <block>

<procedure-heading> → **procedure** <identifier> | **procedure** <identifier> **()** | **procedure** <identifier> **(** <parameter-declaration-list> **)**

<function-declaration> → <function-heading> **;** <block>

<function-heading> → **function** <identifier> **:** <type> | **function** <identifier> **()** **:** <type> | **function** <identifier> **(** <parameter-declaration-list> **)** **:** <type>

<parameter-declaration-list> → <parameter-declaration> | <parameter-declaration> **,** <parameter-declaration-list>

<parameter-declaration> → <identifier-list> **:** <type>

<statement-block> → <statement> | <multiple-statement>

<multiple-statement> → **begin** <statement-list> **end**

<statement-list> → <statement> | <statement> **;** <statement-list>

<statement> → <simple-statement> | <structured-statement>

<simple-statement> → <assignment-statement> | <call-procedure-or-function> | <call-write-read-procedure>

<structured-statement> → <conditional-statement> | <repetitive-statement>

<assignment-statement> → <identifier> **:=** <expression>

<call-procedure-or-function> → <identifier> | <identifier> **()** | <identifier> **(** <expression-list> **)**

<call-write-read-procedure> → **write** | **read** | **write** **()** | **read** **()** | **write** **(** <expression-list> **)** | **read** **(** <expression-list> **)**

<conditional-statement> → **if** <expression> **then** <statament-block> | **if** <expression> **then** <statament-block> **else** <statament-block>

<repetitive-statement> → **while** <expression> **do** <statement-block>

<expression-list> → <expression> | <expression> **,** <expression-list>

<expression> → <expression> **or** <expression$_1$> | <expression$_1$>

<expression$_1$> → <expression$_1$> **and** <expression$_2$>| <expression$_2$>

<expression$_2$> → <expression$_2$> <relational-operator> <expression$_3$> | <expression$_3$>

<expression$_3$> → <expression$_3$> <addition-operator> <expression$_4$> | <expression$_4$>

<expression$_4$> → <expression$_4$> <multiplication-operator> <expression$_5$> | <expression$_5$>

<expression$_5$> → <identifier> | **(** <expression> **)** | <call-procedure-or-function> | <unary-operator> <expression$_5$> | <literal>

<relational-operator> → = | <> | < | <= | > | >=

<unary-operator> → − | **not**

<addition-operator> → + | −

<multiplication-operator> → ∗ | /

<type> → **integer** | **boolean**

<identifier-list> → <identifier> | <identifier> **,** <identifier-list>

<identifier> → <letter> <identifier$_1$> | _ <identifier$_1$>

<identifier$_1$> → <word> <identifier$_1$> | <number> <identifier$_1$> | λ

<literal> → <bool> | <number>

<number> → <digit> <number$_1$>

<number$_1$> → <digit> <number$_1$> | λ

<word> → <letter> <word$_1$> | _ <word$_1$>

<word$_1$> → _ <word$_1$> | <letter> <word$_1$> | λ

<letter> → **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**

<digit> → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<bool> → **true** | **false**

2.7. Problemas encontrados

Como se ha decidido utilizar **BNF** sin extender, fue necesario reemplazar las notaciones repetitivas, optativas y opcionales, que se tuvieron en cuenta en un principio, por una versión equivalente sin ellas. No se han tenido mayores complicaciones, dada la facilidad y simplicidad que otorga la notación de gramáticas.

2.8. Conclusiones

Se ha logrado realizar una gramática en notación **BNF** sin extender, que cumpla con los requisitos del problema [2.2], denotando las limitaciones [2.5] que tendrá el lenguaje que se compilará y considerando optimizar la gramática de manera que se obtenga un mayor rendimiento en la implementación futura [2.4].

Capítulo 3

Especificación léxica

3.1. Introducción

En este capítulo utilizaremos la gramática desarrollada en el capítulo anterior y realizaremos el diseño de un analizador léxico. El objetivo es recorrer la cadena de entrada carácter por carácter y detectar los **tokens** especificados, en este caso Pascal reducido (definido por la gramática). Para ello, se indicarán como están formados los *tokens*, que son elementos que utilizará el analizador sintáctico posteriormente, y se mostrará el diseño de un autómata que permita el reconocimiento de tales tokens en un recorrido [3.6].

3.2. Descripción del problema

Para esta etapa requerimos especificar en un principio el alfabeto que tendrá nuestra cadena de entrada, luego debemos definir que combinación de símbolos corresponderán a lexemas de nuestro lenguaje y partir de estos debemos generar patrones que identifiquen para cada token que lexemas les corresponden [3.3].

Dados los tokens y sus patrones representativos, se especifica un autómata finito determinístico que lee cada símbolo de la cadena de entrada y permite determinar si es un token válido del lenguaje. Para ello el autómata se realiza mediante la unión de autómatas más simples para reconocer los patrones de cada token por separado.

3.3. Definición del alfabeto, lexemas, tokens y patrones

El alfabeto estará compuesto por las letras del abecedario (A-Z y a-z) en conjunción con los dígitos de 0-9 y algunos caracteres especiales:

$$\Sigma = \{\boldsymbol{A}, ..., \boldsymbol{Z}, \boldsymbol{a}, ..., \boldsymbol{z}, \boldsymbol{0}, ..., \boldsymbol{9}, \_, \boldsymbol{\}}, \boldsymbol{\{},), (, ;, \cdot, ,, :, =, <, >, +, -, *, /\}$$

Con estos símbolos podemos definir los lexemas que componen el subconjunto de lenguaje Pascal que se desarrolla y luego definir expresiones regulares para poder identificar a que token corresponden.

En la tabla 3.1 podemos visualizar los token que se indentifican en el lenguaje especificado junto con los patrones que definen los lexemas que se corresponden con cada uno de estos token. Tales patrones se especificarán mediante expresiones regulares.

Token	Patrón	Ejemplo
tk_type_int	*integer*	*integer*
tk_type_bool	*boolean*	*boolean*
tk_boolean_true	*true*	*true*
tk_boolean_false	$false$	$false$
tk_number	$[0-9]^+$	100
tk_id	$([A-Z]\|[a-z]\|\_)([A-Z]\|[a-z]\|\_\|[0-9])^*$	$un\_identificador\_1$
tk_assign	:=	:=
tk_rel_op_eq	=	=
tk_rel_op_neq	<>	<>
tk_rel_op_min	<	<
tk_rel_op_may	>	>
tk_rel_op_leq	<=	<=
tk_rel_op_geq	>=	>=
tk_add_op_sum	+	+
tk_add_op_rest	−	−
tk_mult_op_por	∗	∗
tk_mult_op_div	/	/
tk_bool_op_and	*and*	*and*
tk_bool_op_or	*or*	*or*
tk_not_op	*not*	*not*
tk_if	if	if
tk_then	*then*	*then*
tk_else	*else*	*else*
tk_while	*while*	*while*
tk_do	*do*	*do*
tk_program	*program*	*program*
tk_begin	*begin*	*begin*
tk_end	*end*	*end*
tk_var	*var*	*var*
tk_procedure	*procedure*	*procedure*
tk_function	$function$	$function$
tk_read	*read*	*read*
tk_write	*write*	*write*
tk_opar	(	(
tk_cpar	)	)
tk_tpoints	:	:
tk_endstnc	;	;
tk_point	.	.
tk_comma	,	,

Tabla 3.1: Define para cada token el patrón que identifica los lexemas que le corresponden.

3.4. Estrategias

Utilizamos diferentes tokens para cada uno de los operadores, como en los operadores de comparación, de multiplicación, y de suma. Así aunque se manejen mas cantidad de tokens que si estuvieren agrupados, es mas sencillo diferenciarlos en las etapas posteriores.

Para componer el reconocedor, tomaremos cualquier cadena de entrada que pueda ser una palabra clave, con el patrón del token id. Si es una palabra clave, se crea su token correspondiente, si no lo es, entonces se confirma que es un id. De esta manera se simplifica el autómata reconocedor de tokens.

3.5. Limitaciones

En la especificación de tokens se ha indicado para cada operador un token diferente, como con los operadores relacionales (rel_op), esto permite diferenciarlos y en futuras etapas facilita la implementación. En cambio se pudo optar por una visión mas genérica que facilite la visualización de los diferentes token conceptualmente, de manera que queden agrupados y se diferencien por un atributo que indique específicamente que lexema representa.

Por otro lado, con respecto al diseño del autómata, se consideró, para simplificar el diseño, que las palabras reservadas no serán reconocidas por estados independientes si no que serán un caso específico del patrón para reconocer identificadores (tk_id).

3.6. Diseño del reconocedor

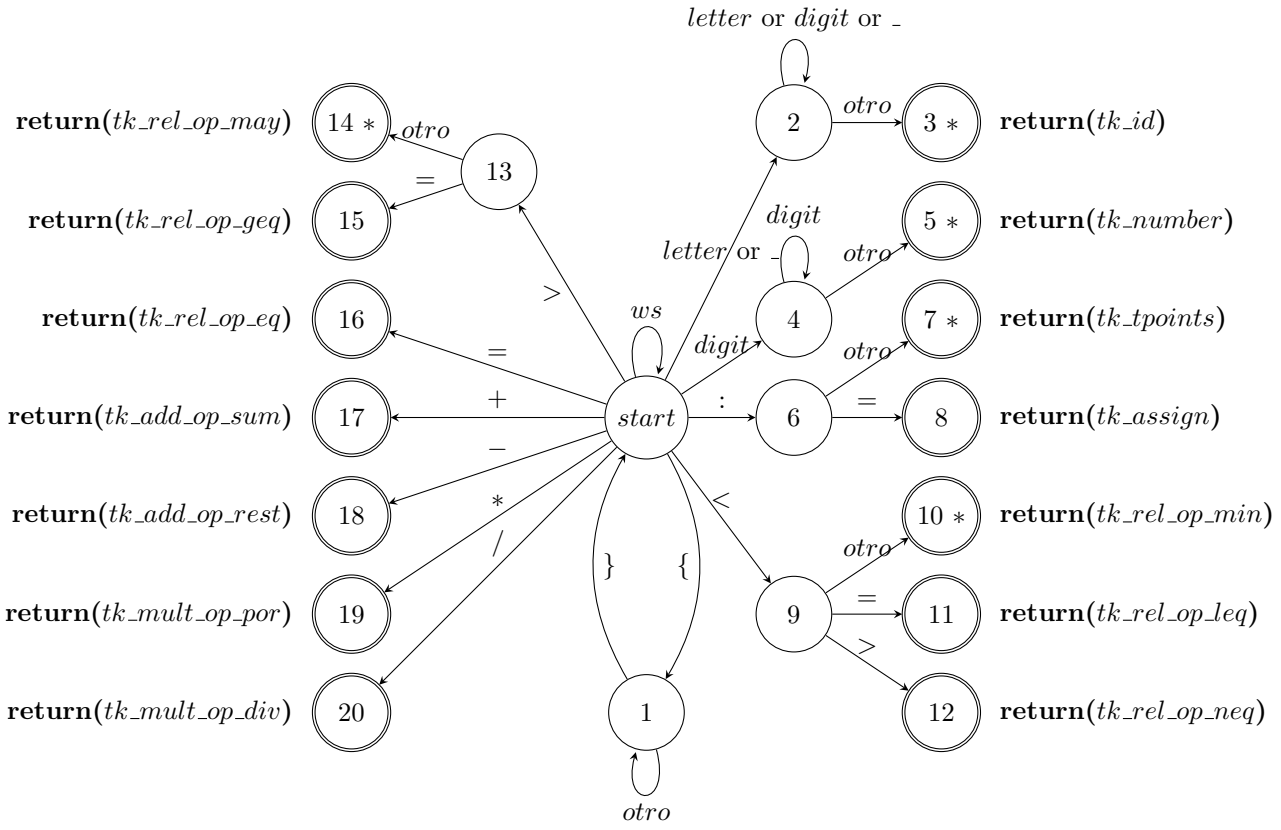


Figura 3.1: Autómata reconocedor de tokens.

Podemos contemplar que en ciertos estados se consume un elemento *otro*, el cuál dependerá de tal estado, y simboliza un elemento que no corresponde con los símbolos cuyos lexemas están incluidos en el patrón del token que está reconociendo. Por ejemplo, en el arco del estado 2-3, *otro* corresponde a símbolos que no son *letter*, *digit* o _. En el caso del arco 4-5 corresponde un símbolo que no es *digit*. En 9-10 para cualquier símbolo que no es = o > y en 13-15 para cualquiera que no sea =. Finalmente en el arco del estado 1, se utiliza de manera que reconozca cualquier símbolo, ya que este es el encargado de reconocer las secciones comentadas del código, lo que permite ignorar cualquier símbolo encerrado entre llaves.

Por otro lado, encontramos el arco etiquetado con *ws* que permite ignorar los espacios, los saltos de línea y las tabulaciones.

A modo de simplificación del gráfico, tanto por cuestiones de tamaño como de estética, no se indicó el reconocimiento de los siguientes tokens: tk_opar, tk_cpar, tk_tpoints, tk_endstnc, tk_point, tk_comma. Estos patrones son triviales, y serían análogos, por ejemplo, al reconocimiento de tk_mult_op.

3.7. Implementación del aplicativo

En esta sección proponemos una implementación de un analizador léxico que se corresponde con la especificación léxica de la sección anterior.

Se dispondrá el enlace del código fuente alojado en GitHub[1]. Durante el desarrollo de este trabajo se ampliará el aplicativo con las demás etapas de análisis sintáctico y semántico. De esta manera, esperamos concluir todas las etapas del compilador para el lenguaje elegido.

3.7.1. Descripción del problema

La idea es desarrollar y documentar un programa capaz de reconocer lexemas de un código fuente basado en la gramática de la sección 2.6, y devolver sus tokens asociados en la tabla 3.1. Para esto, usaremos el autómata de la figura 3.1.

3.7.2. Herramientas utilizadas

Para desarrollar el programa escogimos el lenguaje Java. La versión de Java sobre la que trabajamos es la 8[2](ocho), sobre el sistema operativo Windows 10.

El código de fuente alojado en GitHub tiene la estructura de un proyecto de NetBeans, por lo que puede usarse ese IDE para levantar el proyecto.

3.7.3. Diseño

Para llevar a cabo lo propuesto en este capítulo, y luego poder continuar con el mismo proyecto ampliando el código con las siguientes etapas de análisis, propusimos estructurar el proyecto en *packages*. La estructura del proyecto se puede ver en la figura 3.2, por lo que ahora nos centraremos en trabajar sobre el paquete *lexico*, el cual contiene todas las clases que se usarán en el análisis léxico.

```
src
└── compiladorpascal
    ├── CompiladorPascal.java
    ├── lexico
    ├── sintactico
    └── semantico
```

Figura 3.2: Árbol de directorios del proyecto de Java.

La clase **CompiladorPascal** será la clase principal del proyecto, donde se crearan los objetos necesarios en cada etapa del análisis.

Las clases que se usarán en el análisis léxico son: **AnalizadorLexico, Token, Tokens**. El árbol de directorios actualizado queda como el de la figura 3.3:

```
src
└── compiladorpascal
    ├── CompiladorPascal.java
    ├── lexico
    │   ├── AnalizadorLexico.java
    │   ├── Token.java
    │   └── Tokens.java
    ├── sintactico
    └── semantico
```

Figura 3.3: Árbol de directorios del proyecto de Java con los archivos del analizador léxico.

Descripción de las clases:

[1] https://github.com/Martinnqn/CompiladorPascal

[2] Actualización 171, al día 14/05/2018

- **AnalizadorLexico**: contiene el código necesario para reconocer cada token y reportar los errores léxicos en caso de haberlos. El código tiene una relación directa con el autómata de la figura 3.1.
- **Token**: contiene los atributos que representan un token, tal como su nombre y valor.
- **Tokens**: contiene métodos estáticos que almacenan en HashMaps los nombres y valores de los tokens como palabras reservadas y símbolos. Se utilizará en el AnalizadorLexico para obtener los nombres y patrones de los token del lenguaje.

3.7.4. Instructivos de instalación y uso

Al ser un programa hecho en Java, cuenta con la capacidad de ser portable, por lo que no requiere instalación.

El proceso de compilación puede realizarse mediante un IDE (NetBeans en nuestro caso), o mediante la consola o terminal, con el comando *javac *.java* sobre los archivos del paquete.

3.7.5. Ejemplos

En la figura 3.4 vemos un programa en nuestro Pascal que es léxicamente correcto, por lo que la salida en la figura 3.5 nos muestra una cadena con los tokens obtenidos tras el análisis y no menciona ningún error encontrado. Por otro lado en las figura 3.6 encontramos un código con un caracter desconocido "%" y en la figura 3.7 la correspondiente salida con el error, indicando la línea y posición donde ocurrió y más abajo los tokens que reconoció hasta alcanzar el error. También podemos ver en la figura 3.8 un código donde no se cierra el comentario por lo que en la figura 3.9 vemos que la salida es un error específico para este caso.

```
1  Program Example1;
2  Var
3      Num1, Num2, Sum : Integer;
4      Result: Boolean;
5  Begin {no semicolon}
6      Sum := Num1 + Num2;
7      if (Num1 > Num2) then
8          Result := true
9  End.
```

Figura 3.4: Programa en Pascal reducido léxicamente correcto.

```
run:
<TK_PROGRAM><TK_ID><TK_ENDSTNC><TK_VAR><TK_ID><TK_COMMA>
<TK_ID><TK_COMMA><TK_ID><TK_TPOINTS><TK_TYPE_INT><TK_ENDSTNC>
<TK_ID><TK_TPOINTS><TK_TYPE_BOOL><TK_ENDSTNC><TK_BEGIN><TK_ID>
<TK_ASSIGN><TK_ID><TK_ADD_OP_SUM><TK_ID><TK_ENDSTNC><TK_IF>
<TK_OPAR><TK_ID><TK_REL_OP_MAY><TK_ID><TK_CPAR><TK_THEN>
<TK_ID><TK_ASSIGN><TK_BOOLEAN_TRUE><TK_END><TK_POINT>
```

Figura 3.5: Salida de la ejecución del analizador léxico con el código de la figura 3.4.

```
1  Program Example2;
2  Var
3      %Num1, Num2, Sum : Integer;
4      Result: Boolean;
5  Begin [ {no semicolon}
6      Sum := Num1 ++ Num2;
7      if (Num1 > Num2) then
8          Result := true
9  ]
10 End.
```

Figura 3.6: Programa en Pascal con error léxico por caracter desconocido.

```
run:
Error linea 3 posicion 5. Caracter '%' desconocido.
<TK_PROGRAM><TK_ID><TK_ENDSTNC><TK_VAR>
```

Figura 3.7: Salida de la ejecución del analizador léxico con el código de la figura 3.6.

```
1  Program Example3;
2  Var
3      Num1, Num2, Sum : Integer;
4      Result: Boolean;
5  Begin {no semicolon}
6      {Sum := Num1 + Num2;
7      if (Num1 > Num2) then
8          Result := true
9  End.
```

Figura 3.8: Programa en Pascal con error léxico por no encontrar final de comentario.

```
run:
Error fin de comentario no encontrado.
<TK_PROGRAM><TK_ID><TK_ENDSTNC><TK_VAR><TK_ID><TK_COMMA>
<TK_ID><TK_COMMA><TK_ID><TK_TPOINTS><TK_TYPE_INT><TK_ENDSTNC>
<TK_ID><TK_TPOINTS><TK_TYPE_BOOL><TK_ENDSTNC><TK_BEGIN>
```

Figura 3.9: Salida de la ejecución del analizador léxico con el código de la figura 3.8.

3.8. Conclusiones

En este capítulo pudimos concluir la especificación e implementación del análisis léxico.

Realizamos una definición de los token en la tabla 3.1, considerando cada token conceptualmente como se menciona en 3.5 y así evitamos una definición exhaustiva y simplificamos la tabla.

En base a la tabla de tokens, diseñamos el autómata reconocedor de tokens [3.6] describiendo su funcionamiento.

Luego, en la sección 3.7, tuvimos en cuenta las especificaciones y desarrollamos un programa en Java que permite reconocer los tokens de nuestro lenguaje.

En los siguientes capítulos continuaremos desarrollando las especificaciones e implementaciones de las siguientes etapas de un compilador.

Capítulo 4

Especificación sintáctica

4.1. Introducción

El objetivo de este capítulo es realizar la especificación sintáctica del lenguaje y mostrar un pseudocódigo de su implementación.

Para llevar acabo la especificación sintáctica, trabajaremos sobre algunos aspectos de la gramática de la sección 2.6, realizando modificaciones que permitan implementar un analizador sintáctico descendente predictivo recursivo sin backtracing.

Luego se hará la implementación en lenguaje Java, haciendo uso del Analizador Léxico del capítulo anterior.

4.2. Descripción del problema

Se requiere realizar una modificación sobre la gramática del lenguaje dada en los capítulos anteriores de manera que se obtenga una gramática equivalente que pueda ser utilizada para desarrollar un analizador sintáctico descendente predictivo recursivo. Para ello, la gramática resultante debe estar factorizada a izquierda, no tener recursividad a izquierda y no debe ser ambigua.

Una vez obtenida la gramática y tratados los posibles conflictos se debe proceder en el diseño de una analizador sintáctico que permite evaluar si una cadena de tokens obtenidos mediante el analizador léxico cumple correctamente con la sintaxis dada en la gramática.

4.3. Conflictos de la gramática y el lenguaje

El análisis sintáctico predictivo recursivo es un método top-down de análisis en el que se ejecutan un conjunto de procedimientos recursivos para procesar la entrada.

Para ello se asocia un procedimiento para cada no terminal de la gramática del lenguaje. Al ser un proceso determinístico, se requiere que la gramática a implementar cumpla con ciertas características para evitar conflictos en el desarrollo del analizador:

- **Gramática factorizada a izquierda:** exige que solo exista un camino para elegir en un no terminal. Ejemplo de una regla de producción no factorizada a izquierda: <A> $\rightarrow$ <S> α | <S> β.
- **Gramática sin recursión a izquierda:** hay dos tipos de recursión a izquierda que deben evitarse:
  - Cuando se da de manera inmediata en la misma regla de producción:

    $$<A> \rightarrow <A> \alpha \mid \beta$$

  - Cuando se da a partir de varias reglas de producción:

    $$<S> \rightarrow <A> \beta$$
    $$<A> \rightarrow <S> \alpha \mid \alpha$$

- **Gramática no ambigua:** que exista un único árbol de derivación para una entrada.

Las gramáticas que cumplen con estas características son clasificadas en las clases de gramática LL(1). A continuación trataremos de modificar nuestra gramática para que alcance la clase de gramáticas LL(1) o resuelvan estos conflictos mediante un tratamiento especial.

4.3.1. Factorización a izquierda

Para factorizar a izquierda se utilizó el siguiente algoritmo:

Sea la regla sin factorización a izquierda:

$$<A> \rightarrow \alpha \mid \alpha\ <B>$$

Se procede a reemplazarla por la siguiente construcción, factorizada izquierda, equivalente:

$$<A> \rightarrow \alpha\ <A_1>$$
$$<A_1> \rightarrow <B> \mid \lambda$$

Donde $<A_1>$ es un nuevo no terminal que se genera.

Las reglas a las que se les aplicó la factorización a izquierda se presentan en la lista 4.1 (para simplificar la lista solo pondremos los no terminales del lado izquierdo de la regla):

Lista 4.1: Reglas a las que se le aplica factorización a izquierda.

- <declaration-block>
- <variable-declaration-list>
- <procedure-and-function-declaration-list>
- <procedure-heading>
- <function-heading>
- <parameter-declaration-list>
- <statement-list>
- <call-write-read-procedure>
- <conditional-statement>
- <expression-list>
- <identifier-list>

A modo de ejemplo mostraremos cómo aplicar este método a la regla <statement-list>.

Sea la regla no factorizada a izquierda:

<statement-list> → <statement> | <statement> **;** <statement-list>

El resultado tras factorizarla a izquierda es:

$$<\text{statement-list}> \rightarrow <\text{statement}>\ <\text{statement-list}_1>$$
$$<\text{statement-list}_1> \rightarrow \textbf{;}\ <\text{statement-list}> \mid \lambda.$$

Donde $<\text{statement-list}_1>$ es el nuevo no terminal que se genera con la aplicación del método.

Esto fue aplicado a todas las reglas descritas en la lista 4.1.

4.3.2. Quitando recursión a izquierda

Para eliminar la recursión a izquierda se utilizó el siguiente método:

Sea la regla con recursión a izquierda inmediata:

$$<A> \rightarrow <A>\ \alpha \mid \beta$$

Se procede a reemplazarla por la siguiente construcción, sin recursión a izquierda, equivalente:

$$<A> \rightarrow \beta\ <A_1>$$
$$<A_1> \rightarrow \alpha\ <A_1> \mid \lambda$$

Donde $<A_1>$ es un nuevo no terminal.

Las reglas que fueron modificadas para eliminar la recursión a izquierda se presentan en la lista 4.2 (para simplificar la lista solo pondremos los no terminales del lado izquierdo de la regla):

Lista 4.2: Reglas a las que se le elimina la recursión a izquierda.

- <expression>.
- <expression$_1$>.
- <expression$_2$>.
- <expression$_3$>.
- <expression$_4$>.
- <expression$_5$>.

A modo de ejemplo mostraremos cómo aplicar este método a la regla principal desde la cual comienzan a generarse las expresiones. Como se generan muchos no terminales adicionales, primero cambiamos los nombres para que sean más significativos:

- <expression> fue cambiado por <expression-or>
- <expression$_1$> fue cambiado por <expression-and>
- <expression$_2$> fue cambiado por <expression-rel>
- <expression$_3$> fue cambiado por <expression-add>
- <expression$_4$> fue cambiado por <expression-mult>
- <expression$_5$> fue cambiado por <factor>

Ahora mostraremos cómo es el resultado de aplicar el método:

Sea la regla con recursividad a izquierda:

<expression-or> → <expression-or> “or” <expression-and> | <expression-and>

El resultado tras quitar la recursión a izquierda es:

<expression-or> → <expression-and> <expression-or$_1$>
<expression-or$_1$> → “or” <expression-and> <expression-or$_1$> | λ.

Donde <expression-or$_1$> es el nuevo no terminal que se genera con la aplicación del método.

Esto fue aplicado a todas las reglas descritas en la lista 4.2

4.3.3. Ambigüedad del lenguaje

La ambigüedad surge cuando existe más de una forma de derivar una cadena. En este lenguaje, cuando se generan construcciones que contienen `if-then-else` anidados, se pueden generar dos árboles diferentes de derivación. En las figuras 4.2 y 4.3 se ven los árboles posibles para la derivación del siguiente fragmento de programa en nuestro lenguaje:

```
1  if (cond_1) then
2          if (cond_2) then
3                  code_then
4          else
5                  code_else
```

Figura 4.1: Fragmento de código con `if` anidados.

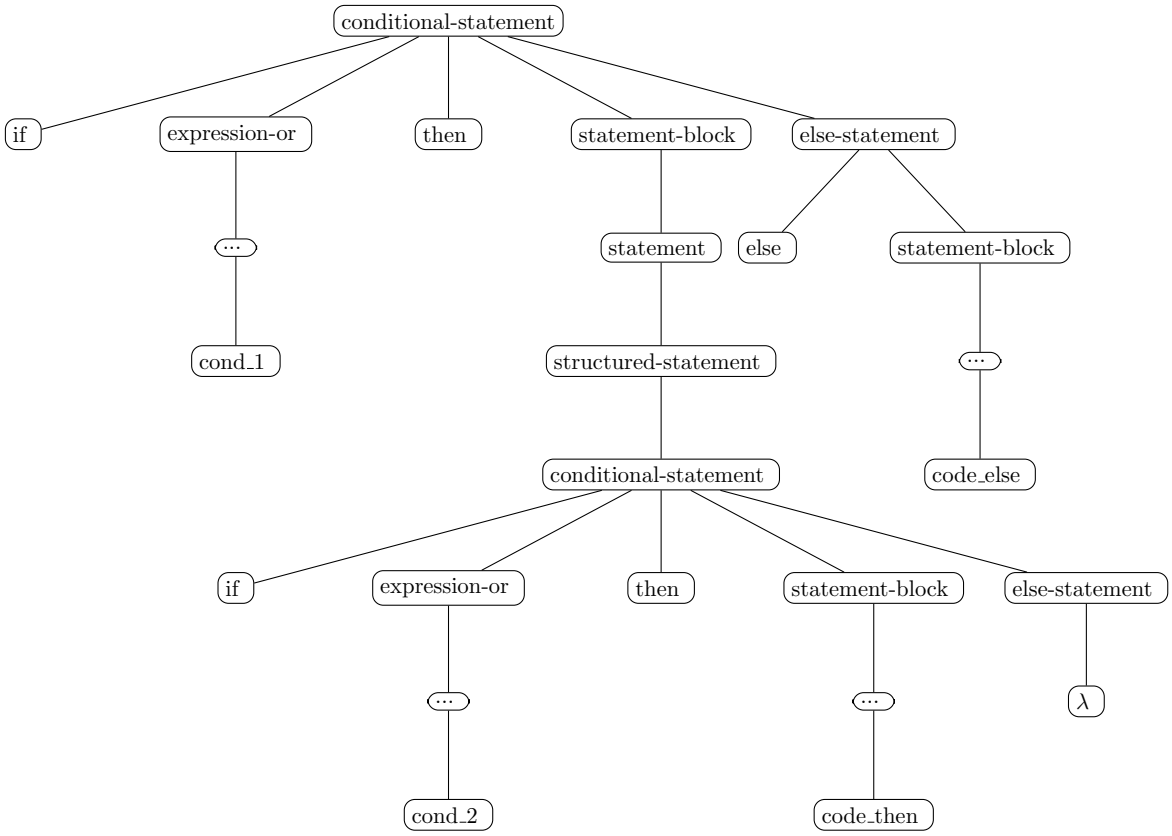


Figura 4.2: Árbol de derivación A para la construcción `if-then-else`.

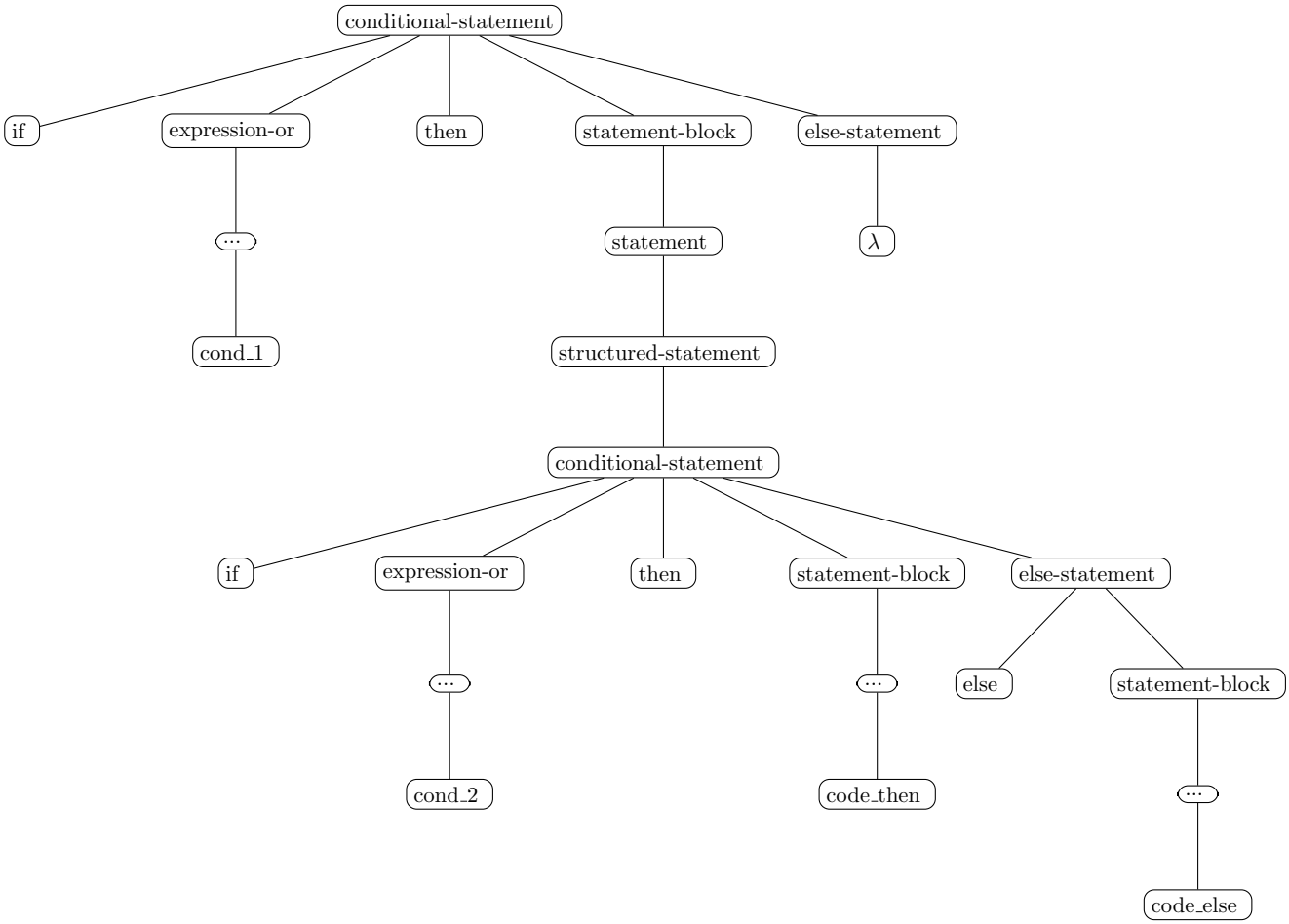


Figura 4.3: Árbol de derivación B para la construcción `if-then-else`.

El problema surge al momento de decidir si el código correspondiente al `else` se corresponde con el `if` interno o el externo. Este inconveniente no permite que la gramática alcance la clase de gramáticas LL(1). Sin embargo, se puede forzar la implementación del analizador sintáctico para que tome por un camino específico cuando se encuentra esta construcción. La forma más natural es la de la figura 4.3, donde se asocia el `else` al `if` más interno en la estructura de `if` anidados. Esta es la forma que adoptaremos en la implementación de nuestro analizador.

4.4. Gramática modificada

En esta sección se muestra la gramática con las modificaciones que fueron posibles realizar.

Se eliminó la recursión a izquierda inmediata y también se factorizó a izquierda.

Además se eliminó un problema que surgía en el no terminal <expression$_5$>, ya que desde este se podía derivar en <identifier> y en <call-procedure-or-function>, y este último también derivaba en <identifier>, lo que ocasionaba problemas de factorización a izquierda.

El único inconveniente es que no es posible eliminar la ambigüedad, ya que esta es inherente al lenguaje que está siendo especificado. En la sección 4.3.3 se mostró dónde surge la ambigüedad.

<program> → <program-heading> <block> **.**

<program-heading> → **program** <identifier> **;**

<block> → <declaration-block> <multiple-statement> | <multiple-statement>

<declaration-block> → <variable-declaration-block> <declaration-block$_1$> | <declaration-block$_1$>

<declaration-block$_1$> → <procedure-and-function-declaration-list> | λ

<variable-declaration-block> → **var** <variable-declaration-list>

<variable-declaration-list> → <variable-declaration> **;** <variable-declaration-list$_1$>

<variable-declaration-list$_1$> → <variable-declaration-list> | λ

<variable-declaration> → <identifier-list> **:** <type>

<procedure-and-function-declaration-list> → <procedure-declaration> **;** <procedure-and-function-declaration-list$_1$> | <function-declaration> **;** <procedure-and-function-declaration-list$_1$>

<procedure-and-function-declaration-list$_1$> → <procedure-and-function-declaration-list> | λ

<procedure-declaration> → <procedure-heading> **;** <block>

<procedure-heading> → **procedure** <identifier> <parameters>

<function-declaration> → <function-heading> **;** <block>

<function-heading> → **function** <identifier> <parameters> **:** <type>

<parameters> → **(** <parameters$_1$> **)** | λ

<parameters$_1$> → <parameter-declaration-list> | λ

<parameter-declaration-list> → <parameter-declaration> <parameter-declaration-list$_1$>

<parameter-declaration-list$_1$> → **,** <parameter-declaration-list> | λ

<parameter-declaration> → <identifier-list> **:** <type>

<statement-block> → <statement> | <multiple-statament>

<multiple-statement> → **begin** <statement-list> **end**

<statement-list> → <statement> <statement-list$_1$>

<statement-list$_1$> → **;** <statement-list> | λ

<statement> → <simple-statement> | <structured-statement>

<simple-statement> → <identifier> <simple-statement$_1$> | **write** <call-procedure-or-function> | **read** <call-procedure-or-function>

<simple-statement$_1$> → <assignment-statement> | <call-procedure-or-function> | λ

<structured-statement> → <conditional-statement> | <repetitive-statement>

<assignment-statement> → **:=** <expression-or>

<call-procedure-or-function> → **(** <call-procedure-or-function$_1$> **)**

<call-procedure-or-function$_1$> → <expression-list> | λ

<conditional-statement> → **if** <expression-or> **then** <statament-block> <else-statement>

<else-statement> → **else** <statement-block> | λ

<repetitive-statement> → **while** <expression-or> **do** <statament-block>

<expression-list> → <expression-or> <expression-list$_1$>

<expression-list$_1$> → **,** <expression-list> | λ

<expression-or> → <expression-and> <expression-or$_1$>

<expression-or$_1$> → **or** <expression-and> <expression-or$_1$> | λ

<expression-and> → <expression-rel> <expression-and$_1$>

<expression-and$_1$> → **and** <expression-rel> <expression-and$_1$> | λ

<expression-rel> → <expression-add> <expression-rel$_1$>

<expression-rel$_1$> → <relational-operator> <expression-add> <expression-rel$_1$> | λ

<expression-add> → <expression-mult> <expression-add$_1$>

<expression-add$_1$> → <addition-operator> <expression-mult> <expression-add$_1$> | λ

<expression-mult> → <factor> <expression-mult$_1$>

<expression-mult$_1$> → <multiplication-operator> <factor> <expression-mult$_1$> | λ

<factor> → <identifier> <factor$_1$> | **(** <expression-or> **)** | <unary-operator> <factor> | <literal>

<factor$_1$> → <call-procedure-or-function> | λ

<relational-operator> → = | <> | < | <= | > | >=

<unary-operator> → − | **not**

<addition-operator> → + | −

<multiplication-operator> → ∗ | /

<type> → **integer** | **boolean**

<identifier-list> → <identifier> <identifier-list$_1$>

<identifier-list$_1$> → **,** <identifier-list> | λ

<identifier> → <letter> <identifier$_1$> | _ <identifier$_1$>

<identifier$_1$> → <word> <identifier$_1$> | <number> <identifier$_1$> | λ

<literal> → <bool> | <number>

<number> → <digit> <number$_1$>

<number$_1$> → <digit> <number$_1$> | λ

<word> → <letter> <word$_1$> | _ <word$_1$>

<word$_1$> → _ <word$_1$> | <letter> <word$_1$> | λ

<letter> → **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**

<digit> → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

<bool> → **true** | **false**

4.5. Estrategias

Al tener que realizar muchos cambios sobre la gramática, era probable cometer errores, por lo que primero decidimos cambiar los nombres de algunos no terminales para que sean mas expresivos, y luego aplicamos la factorización a izquierda y después la eliminación de la recursión a izquierda. Estos dos algoritmos podrían haberse realizado en el orden inverso, pero ejecutados en el orden que propusimos nos aseguramos que luego de eliminar la recursividad no se generaran construcciones que no estén factorizadas a izquierda.

En cuanto a la ambigüedad del lenguaje, como dijimos en 4.3.3, tomamos como criterio hacer corresponder el `else` con el `if` más interno.

4.6. Limitaciones

La única limitación encontrada fue la expresividad del lenguaje, que hace que algunas construcciones sean ambiguas. Sin embargo no presenta un gran problema, ya que tomamos decisiones que permiten construir el analizador sintáctico descendente. En la siguiente sección mostraremos cómo esta limitación puede ser mitigada.

4.7. Diseño del analizador sintáctico

En el apéndice A.1 se expone el pseudocódigo del analizador sintáctico descendente predictivo recursivo basado en la grámatica modificada dada en la sección 4.4. Este dispone de un método `match` que permite verificar si un terminal de la gramática coincide con el símbolo de preanálisis obtenido a partir del analizador léxico. Por otro lado dispone de un método para cada no terminal de la gramática, donde se evalúa sus conjuntos de Primeros y se determina con qué terminales debe unificar o a qué otros no terminales debe llamar. En caso de no coincidir tales conjuntos de primeros, se procede a lanzar un `error`, a excepción de los casos donde la gramática acepta λ como opción de la producción.

4.8. Implementación del aplicativo

Para la implementación del analizador sintáctico, utilizaremos el pseudocódigo del apéndice A.1 y mostraremos su implementación en lenguaje Java. Para ello se explicarán algunos procedimientos de ejemplo, y luego el resto de estos serán análogos.

El Analizador Sintáctico trabaja en conjunto con el Analizador Léxico, recibiendo como entrada los tokens que reconozca y devuelva el Analizador Léxico.

Todo lo desarrollado se puede encontrar en el mismo proyecto de GitHub donde está el Analizador Léxico.

4.8.1. Descripción del problema

Para esta etapa requerimos implementar el analizador sintáctico descendente predictivo recursivo, cuya tarea es verificar si la sintaxis de un código fuente cumple las construcciones gramaticales que propone nuestra gramática modificada de la sección 4.4 y que se corresponda con el pseudocódigo dado en la sección 4.7.

En caso de que haya un token que no pertenezca a alguna construcción válida, lanzaremos un error adecuado.

Luego de concluir la implementación, mostraremos ejemplos de ejecuciones exitosas y fallidas, como así también las instrucciones para poder utilizar el Analizador Sintáctico.

4.8.2. Herramientas utilizadas

Para desarrollar el programa continuamos desde el mismo proyecto que el Analizador Léxico, en lenguaje Java. La versión de Java sobre la que trabajamos es la 8[1](ocho), sobre el sistema operativo Windows 10.

Continuamos con el proyecto de NetBeans que se encuentra en GitHub.

[1] Actualización 171, al día 14/05/2018

4.8.3. Diseño

La idea del Analizador Sintáctico es crear un procedimiento para cada no terminal de la gramática, que se encargue de verificar si la sucesión de caracteres de la entrada cumple con la especificación del lenguaje. Al utilizar el analizador léxico, evitaremos recorrer caracter por caracter, y nos centraremos en los tokens recibidos del Analizador Léxico.

En este sentido expandiremos el proyecto que disponiamos anteriormente, agregando una nueva clase **AnalizadorSintactico**, como se ve en la figura 4.4 que dispondrá de los métodos necesarios para llevar a cabo el análisis sintáctico.

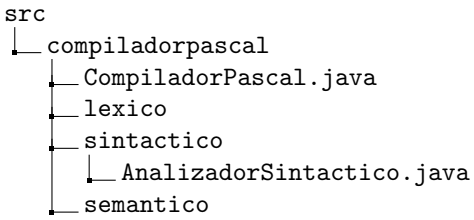


Figura 4.4: Árbol de directorios del proyecto de Java con los archivos del analizador sintáctico.

En el código del AnalizadorSintactico el procedimiento más importante es el `match`, que verifica si un token recibido de la entrada (el String `preanalisis`), es igual al token que espera el analizador sintáctico (String `terminal`). Si son iguales puede continuar avanzando sobre la entrada, y si no son iguales hay un error sintáctico. En la figura 4.5 puede verse el código del método `match`.

```
1  private void match(String terminal) {
2          if (preanalisis.getNombre().equals(terminal)) {
3                  preanalisis = lexico.tokenSiguiente();
4          } else {
5                  error(terminal);
6          }
7  }
```

Figura 4.5: Método `match` implementado en Java.

Existen dos tipos de errores, ambos lanzan un error en tiempo de ejecución, donde muestran la línea y posición del caracter donde ocurre el error, además del token que recibieron desde la entrada el cual no coincide con la gramática. Uno de los métodos de error se muestra a continuación en la figura 4.6, el cual recibe por parámetro el terminal, para lanzar un error más específico avisando cuál es el token que se esperaba.

```
1  private void error(String terminal) {
2          throw new RuntimeException("sintactico", new Throwable(
3                          "\nError sintactico: linea " + lexico.getNroLinea()
4                          + " posicion " + (lexico.getPos() + 1) + ".\nSimbolo de preanalisis "
5                          + preanalisis.getNombre() + " no esperado. Se esperaba " + terminal));
6  }
```

Figura 4.6: Método `error` implementado en Java para lanzar errores sintácticos.

Para ver el funcionamiento del procedimiento `match`, se mostrará cómo se utiliza en el método asociado al no terminal `program_heading`, en el código de la figura 4.7.

```
private void program_heading() {
        if (preanalisis.getNombre().equals("TK_PROGRAM")) {
                match("TK_PROGRAM");
                identifier();
                match("TK_ENDSTNC");
        } else {
                error("TK_PROGRAM");
        }
}
```

Figura 4.7: Método `program_heading` implementado en Java.

Antes de analizar el algoritmo es necesario saber que el conjunto Primeros para un no terminal α contiene los terminales que aparecen como los Primeros símbolos de las cadenas que pueden ser derivadas por α. Este concepto es importante para entender la explicación del algoritmo.

Lo que está simulando este algoritmo es la estructura requerida al comienzo de un programa: **program** `identificador ;`. Primero verifica el conjunto Primeros del no terminal `program_heading`, si el token `preanalisis` coincide (en este caso solo hay un elemento en el conjunto Primeros) entonces llama a `match` con el token `TK_PROGRAM`, luego continúa llamando al procedimiento `indentifier()`, que se encarga de verificar si lo que continúa en la entrada es un `TK_ID` y por último verifica que haya un punto y coma al final de la sentencia, enviando el token `TK_ENDSTNC` al método `match`. Si `preanalisis` no coincide con ningún elemento en el conjunto Primeros, entonces lanza error sintáctico.

En este procedimiento, al tener un único elemento en el conjunto Primeros, es posible llamar al método error que recibe el token que se esperaba, para desplegar un error más expresivo. Si el conjunto Primeros tiene más de un elemento, entonces no es posible dar un mensaje específico de cual es el token que se esperaba, por lo que se imprime un mensaje más genérico.

Para cada no terminal, el procedimiento es análogo a este, con la diferencia de los no terminales que pueden derivar en la cadena nula (aquellos que tienen como opción λ). Para estos no terminales, si el token de preanalisis se corresponde con algunos de los tokens del conjunto Primeros de ese no terminal, entonces realiza el análisis adecuado para ese token, análogo al método `program_heading`; pero si el token no se corresponde con alguno de los tokens de ese no terminal, se deduce que ese no terminal deriva en λ y no se efectúa ningún análisis en ese no terminal, y tampoco se lanza ningún error. A continuación se muestra en la figura 4.8 este comportamiento para el procedimiento `else_statement` asociado al no terminal <else-statement>.

```
private void else_statement() {
        if (preanalisis.getNombre().equals("TK_ELSE")) {
                match("TK_ELSE");
                statement_block();
        }
}
```

Figura 4.8: Método `else_statement` implementado en Java.

Por último hay no terminales que tienen más de una opción para realizar una derivación. Para elegir qué camino tomar, se utiliza un `switch` con los tokens que están en el conjunto Primeros de ese no terminal.

Para mostrar este caso vamos a basarnos en el no terminal `structured-statement`. A partir de `structured-statement` se puede ir hacia `conditional-statement` o `repetitive-statement`. Para decidir qué camino tomar se calcula el conjunto de Primeros de ambos no terminales: Primeros(`conditional-statement`) = {`if`}; Primeros(`repetitive-statement`) = {`while`}. Luego de tener ambos conjuntos, se utilizan en el `switch` para guiar la derivación, como muestra el Algoritmo 4.9.

```
private void structured_statement() {
        switch (preanalisis.getNombre()) {
                case "TK_IF":
                        conditional_statement();
                        break;
                case "TK_WHILE":
                        repetitive_statement();
                        break;
                default:
                        error();
                        break;
        }
}
```

Figura 4.9: Método `structured_statement` implementado en Java.

Estos son los casos básicos para los procedimientos de los no terminales. El procedimiento principal al que se llama para iniciar el análisis sintáctico se muestra en la figura 4.10, el cual es el encargado de capturar los errores lanzados.

```
public void analizar() {
        try {
                preanalisis = lexico.tokenSiguiente();
                program();
        } catch (RuntimeException ex) {
                System.out.println(ex.getCause().getMessage());
        }
}
```

Figura 4.10: Método `analizar` de la clase `AnalizadorSintactico` implementada en Java para comenzar con el analisis.

4.8.4. Instructivos de instalación y uso

Para usar el programa es idéntico que para el Análizador Léxico, al ser un programa portable, solo se requiere la compilación a través del comando *javac *.java* o cargando el proyecto en el NetBeans y usando este para compilar.

Para ejecutar, hay que invocar el programa compilado enviándole como parámetro el archivo con el código fuente del programa a compilar.

4.8.5. Ejemplos

En la figura 4.11 vemos un programa en Pascal que es sintácticamente correcto, por lo que en la salida no retornará ningún tipo de error.

Por otro lado en la figura 4.12 encontramos que la palabra reservada `Program` que indica el comienzo del programa está mal escrita (`Programa`) por lo que el analizar devolverá un error específico, como se muestra en la salida en la figura 4.13, donde indica que se esperaba un `TK_PROGRAM`. , es decir que se esperaba la palabra `Program`, según corresponde en la definición del procedimiento, donde realiza una llamada a `match` con el token `TK_PROGRAM` como parámetro.

También podemos ver en la figura 4.14 otro tipo de error mas general, donde luego de realizar la definición de la variable `Result` se le quiere asignar un tipo de dato `Bool`, el cual no es un tipo existente (el correcto sería `Boolean`). En este sentido, el procedimiento que evalúa la sintaxis de las definiciones de variables, no puede saber que token específico debería matchearse en tal lugar, pero en cambio sabe que corresponde a un "tipo de dato" debido a que el error aparecerá en la llamada al no terminal `type` el cuál devolverá como parámetro de error que se esperaba un tipo de dato, que en este caso corresponden a `Integer` o `Boolean`.

Por último encontramos un tipo de error mas general que los anteriores, como se ve en la figura 4.16 donde no se especificó que token o elemento sintáctico (como un tipo de dato del caso anterior) se esperaba, y por lo tanto no presenta ningún mensaje adicional de lo que "se esperaba" en la cadena sintáctica donde surgió el error, como puede verse en la figura 4.17 de la salida. Estos errores son detectados y es posible determinar a que elemento sintáctico corresponden, pero aún no se han trabajado en su totalidad en esta primera implementación, por lo que quedan pendientes para futuras implementaciones.

```
1  Program Example1;
2  Var
3      Num1, Num2, Sum : Integer;
4      Result: Boolean;
5  Begin {no semicolon}
6      Sum := Num1 + Num2;
7      if (Num1 > Num2) then
8          Result := true
9  End.
```

Figura 4.11: Programa en Pascal reducido sintácticamente correcto.

```
1  Programa Example2;
2  Var
3      Num1, Num2, Sum : Integer;
4      Result: Boolean;
5  Begin {no semicolon}
6      Sum := Num1 + Num2;
7      if (Num1 > Num2) then
8          Result := true
9  End.
```

Figura 4.12: Programa en Pascal reducido con error sintáctico específico.

```
Error sintactico: linea 1 posicion 9.
Simbolo de preanalisis TK_ID no esperado. Se esperaba TK_PROGRAM
```

Figura 4.13: Salida de la ejecución del analizador sintáctico con el código de la figura 4.12.

```
1  Program Example3;
2  Var
3      Num1, Num2, Sum : Integer;
4      Result: Bool;
5  Begin {no semicolon}
6      Sum := Num1 + Num2;
7      if (Num1 > Num2) then
8          Result := true
9  End.
```

Figura 4.14: Programa en Pascal reducido con error sintáctico general.

```
Error sintactico: linea 4 posicion 17.
Simbolo de preanalisis TK_ID no esperado. Se esperaba un tipo de dato
```

Figura 4.15: Salida de la ejecución del analizador sintáctico con el código de la figura 4.14.

```
1  Program Example4;
2  Vars
3      Num1, Num2, Sum : Integer;
4      Result: Boolean;
5  Begin {no semicolon}
6      Sum := Num1 + Num2;
7      if (Num1 > Num2) then
8          Result := true
9  End.
```

Figura 4.16: Programa en Pascal reducido con error sintáctico general y sin un token o elemento sintáctico esperado.

```
Error sintactico: linea 2 posicion 5.
Simbolo de preanalisis TK_ID no esperado.
```

Figura 4.17: Salida de la ejecución del analizador sintáctico con el código de la figura 4.16.

4.9. Problemas encontrados

Dadas las características del lenguaje, no fue posible alcanzar la clase de gramáticas LL(1), pero esto no impidió que se tomaran decisiones de diseño para poder implementar el Analizador sintáctico.

4.10. Posibles mejoras

Como mejora, en el caso de los mensajes de errores que se muestran en la compilación, debería evitarse mostrar los nombres de los tokens que se utiliza, y en su lugar deberían mostrarse los lexemas o lo que ese token representa, para ocultar aspectos de la implementación y mostrar mensajes más intuitivos.

También deberían indicarse los mensajes de error para los casos faltantes donde "se esperaría" un elemento sintáctico general y no un token específico.

4.11. Conclusiones

Pudimos realizar algunas modificaciones propuestas para la gramática, eliminando su recursividad a izquierda y factorizando a izquierda, pero no pudimos cumplir el requisito de que no sea ambigua, ya que el lenguaje subyacente no lo permitía.

Más allá de no alcanzar la clase de gramáticas LL(1), el diseño y la implementación del analizador sintáctico descendente predictivo recursivo sin backtracking se pudieron concluir con éxito, tomando las decisiones pertinentes.

Como resultado de las modificaciones sobre la gramática, ahora es más extensa y se dificulta su legibilidad y comprensión.

Capítulo 5

Especificación semántica

5.1. Introducción

El objetivo de este capítulo es realizar la especificación semántica del lenguaje y mostrar un pseudocódigo de su implementación.

Para llevar acabo la especificación semántica, trabajaremos sobre el analizador sintáctico de la sección anterior, realizando modificaciones que permitan implementar el analizador semántico.

5.2. Descripción del problema

En esta etapa del desarrollo del compilador necesitamos extender el analizador sintáctico para poder realizar verificaciones semánticas sobre el programa a compilar. Los chequeos semánticos a realizar son:

- Existencia: los identificadores utilizados en el cuerpo del programa deben estar previamente declarados. El alcance de los identificadores es estático, por lo que un identificador no declarado en un ambiente se buscará en sus ambientes ancestros.
- Unicidad: no se permite tener más de un identificador con el mismo nombre declarado en un mismo ambiente.
- Correspondencia de parámetros: la llamada a procedimientos o funciones debe coincidir con la signatura del procedimiento o la función invocada.
- Compatibilidad y equivalencia de tipos: los operandos de un operador deben ser compatibles para poder ejecutar correctamente la operación.

5.3. Compatibilidad de tipos

En la tabla 5.1 se muestra cada operador, los tipos de los operandos que puede recibir y el tipo de resultado de la operación.

El operador "-" es el único operador sobrecargado, ya que puede ser utilizado tanto para operaciones binarias como unarias. En el caso de usarlo como operación unaria, su tipo de operando y resultado es el mismo que para la operación binaria.

Operador	Operando 1	Operando 2	Resultado
+, -,*,/	Integer	Integer	Integer
=, <>	Integer/Boolean	Integer/Boolean	Boolean
<,<=,>,>=	Integer	Integer	Boolean
and, or	Boolean	Boolean	Boolean
not	Boolean	-	Boolean
:=	Boolean	Boolean	-
:=	Integer	Integer	-

Tabla 5.1: Compatibilidad de tipos

5.4. Estrategias

Se utilizaron estructuras de objetos para la representación de la tabla de símbolos de manera que esta fuera fácilmente accedida por el Analizador Semántico. En tal estructura se dispusieron atributos de manera que se pueda identificar el nombre del programa, función o procedimiento que corresponde al ambiente nuevo, como también listas para los tipos de los identificadores reconocidos en su interior, y para los parámetros que recibe, en caso de que corresponda. Esto nos permitió chequear fácilmente el retorno de las funciones y las llamadas recursivas, los cuáles son casos particulares que deben tratarse.

5.5. Diseño del analizador semántico

Para modelar el diseño del analizador semántico, se trabajará sobre algunas reglas de la gramática de la sección 4.4, agregando acciones semánticas. Estas acciones permiten extender la potencia de las gramáticas libres de contexto, convirtiendo la gramática en una gramática de atributos.

Toda aquella información que se requiera persistir para las acciones semánticas, y que no pueda ser enviada mediante parámetros entre las llamadas de las reglas de la gramática, será almacenada en la tabla de símbolos para su posterior acceso.

5.5.1. Tabla de símbolos

La tabla de símbolos (TS) es la estructura que guarda toda la información necesaria para el proceso de análisis semántico. En este trabajo, proponemos una tabla de símbolo para cada ambiente particular. Como el alcance de los identificadores es estático, cada identificador pertenecerá a un ambiente A, y podrá ser accedido por aquellos ambientes A' que estén contenidos en A.

El Analizador Semántico mantendrá siempre una referencia a la Tabla de Símbolos del ambiente actual que esté procesando.

En la Tabla 5.2 se esquematiza la forma que tendrá la tabla de símbolos explicada.

Ambiente	Nombre	Tipo	Listas de valores		
1	$ambiente_1$	[program | function | procedure]	Identificador	Tipo	Parámetros
			id_{var}	[integer | boolean]	[ ]
			$id_{subprogram}$	void	[$param_1$, ..., $param_n$]
			...		
			id_n	...	...
...					
n	$ambiente_n$	...	...	...	...

Tabla 5.2: Esquema de la tabla de símbolos diseñada.

5.5.2. Gramática de atributos

Algunas reglas de la gramática serán alteradas para insertar y recuperar información de la tabla de símbolos, y permitirán cumplir con los objetivos propuestos en la sección 5.2. Los siguientes no terminales se verán

modificados, y sus correspondientes implementaciones en el Analizador Sintáctico cambiaran según estas modificaciones:

- <bool> sintetiza el atributo `type` indicando el type boolean.
- <number> sintetiza el atributo `type` indicando el type integer.
- <literal> sintetiza el atributo `type` con el type del literal. Este valor se extrae del atributo sintetizado de <bool> o <number>.
- <factor> sintetiza el atributo `type`. También verifica que pueda aplicarse el operador unario correctamente en <factor> $\rightarrow$ <factor$_1$> utilizando el atributo `type` de <factor$_1$>.
- <identifier> sintetiza el atributo id con el valor del identificador.
- <type> sintetiza el atributo `type` indicando el valor del tipo de dato.
- <expression-or>, <expression-and>, <expression-rel>, <expression-add>, <expression-mult> y <factor> sintetizan el atributo `type` que indica el tipo dato de la expresión.
- <expression-or$_1$ >, <expression-and$_1$ >, <expression-rel$_1$ >, <expression-add$_1$ >, <expression-mult$_1$ > verifican que pueda aplicarse su operación correspondiente usando el atributo `type1` sintetizado y `type2` heredado .
- <identifier-list> sintetiza el atributo `list_id` que devuelve una lista de identificadores. Para cargar la lista utiliza el atributo sintetizado `id` de <identifier>.
- <variable-declaration> utiliza los atributos sintetizados `list_id` y `type` para insertar en la tabla de símbolos cada identificador con su tipo.
- <parameter-declaration> sintetiza un atributo `list_parameters` que devuelve una lista con listas de identificadores y sus tipos.
- <procedure-heading> utiliza el atributo sintetizado `id` y lo guarda en la tabla de simbolos. También utiliza el atributo sintetizado de <parameter-declaration> para guardar los parámetros y sus tipos en la tabla de símbolos
- <function-heading> utiliza los atributos sintetizados `id` y `type` y los guarda en la tabla de símbolos. También utiliza el atributo sintetizado de <parameter-declaration> para guardar los parámetros y sus tipos en la tabla de símbolos
- <assignment-statement> utiliza el atributo heredado `type` junto con el atributo sintetizado `type` de <expression-or> para verificar si se puede efectuar la asignación.
- <conditional-statement> y <repetitive-statement> utilizan el atributo sintetizado `type` de <expression-or> para verificar que la expresión de la condición sea booleana.

5.6. Implementación del aplicativo

En esta sección mostraremos las principales modificaciones del analizador sintáctico teniendo en cuenta la gramática 5.5.2.

5.6.1. Descripción del problema

En esta etapa requerimos implementar el Analizador Semántico. Como es un análisis que debe hacerse en conjunto con el análisis sintáctico, el código del Analizador Semántico será embebido en el Analizador Sintáctico.

Cuando se encuentre un error semántico, se lanzará un error que brinde información significativa para poder corregirlo.

5.6.2. Herramientas utilizadas

Para desarrollar el programa continuamos desde el mismo proyecto que el Analizador Sintáctico, en lenguaje Java. La versión de Java sobre la que trabajamos es la 8[1](ocho), sobre el sistema operativo Windows 10.

5.6.3. Diseño

La idea del Analizador Semántico es modificar los procedimientos implementados del Analizador Sintáctico. Cuando un procedimiento requiera acceder a atributos sintetizados, creará variables para poder recibir los atributos; y cuando un procedimiento requiera atributos heredados, los recibirá por parámetro.

Para no perder la implementación del Analizador Sintáctico, crearemos otra clase llamada **AnalizadorSemántico** dentro del paquete **semantico** con el mismo código del **AnalizadorSintactico**, y además crearemos el TDA **Ambiente**, por lo que nuestro árbol de directorios quedará como en la figura 5.1.

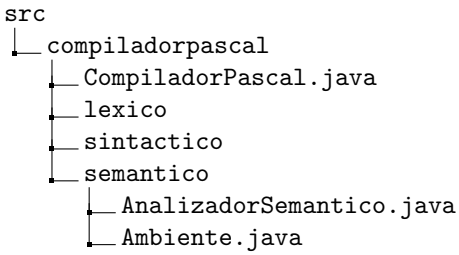


Figura 5.1: Árbol de directorios del proyecto de Java con los archivos del analizador semántico.

Para manejar los errores, creamos los métodos `errorSemantico()` y `errorSintactico()`, para que sea más fácil la invocación de errores, y se puedan mostrar mensajes adecuados.

El diseño de la tabla de símbolos se basa en guardar todo el contenido de un ambiente en su respectiva instancia del TDA **Ambiente**. Para cada ambiente del programa se guardan sus identificadores. Si el identificador es una variable o una función se le asigna su tipo, en el caso de los procedimientos se les asigna el tipo `void` por defecto. Si el identificador es una función o procedimiento, se le asocia una lista que contiene los tipos de sus parámetros manteniendo el orden en el que son declarados.

Cuando inicia el programa se crea el ambiente principal, y luego cada vez que se inicia una función o un procedimiento se crea un ambiente nuevo y se asigna como padre de este al ambiente anterior. Cuando se termina de analizar el cuerpo de un procedimiento o función, se actualiza la referencia del analizador semántico con el padre del ambiente analizado.

A continuación podemos ver la interfaz de la clase **Ambiente**.

```
1  public class Ambiente {
2          Ambiente padre; //ambiente padre
3          //puede ser program, function o procedure
4          private String tipoAmbiente;
5          //nombre del ambiente
6          private String nombre;
7          //Asocia un identificador a su type Void para procedimientos
8          private HashMap<String, String> tipos;
9          //Asocia un nombre de un identificador de funcion o procedimiento a su lista
10         //de parametros (solo el type de los parametros), como <nombreFuncion,
           ↪  parametros>
11         private HashMap<String, LinkedList<String>> parametros;
12
13         public Ambiente(String tipoAmbiente, String nombre, Ambiente padre) {}
14
15         public Ambiente getPadre() {}
16
```

[1]Actualización 171, al día 14/05/2018

```
17          public String getTipoAmbiente() {}
18
19          public void setTipoAmbiente(String tipoAmbiente) { }
20
21          public String getNombre() {}
22
23          public HashMap<String, String> getTipos() {}
24
25          public LinkedList<String> getParametros(String id) {}
26
27          public void setParametros(String id, LinkedList<String> parametros) {}
28
29          public void addParametro(String id, String tipo) {}
30
31          public void addVariable(String id, String tipo) {}
32
33          public void addFunction(String id, String tipo) {}
34
35          public void addProcedure(String id) {}
36
37          public String getTipo(String id) {}
38
39          public boolean equals(String ident, LinkedList<String> param) {}
40
41          public String toString() {}
42  }
```

A continuación se exponen algunos fragmentos de código donde se utilizan las instancias de Ambiente para realizar las verificaciones semánticas.

Declaración de variables: cuando se encuentra una sección de declaración de variables, al insertarlas se verifican que no existan en la tabla de símbolos. Si ya existe una variable en la tabla de símbolos entonces ocurre un error de unicidad.

```
1   private void variable_declaration() {
2           if (preanalisis.getNombre().equals("TK_ID")) {
3                   LinkedList<String> idents = new LinkedList<>();
4                   identifier_list(idents);
5                   match("TK_TPOINTS");
6                   String type = type();
7                   //carga los identificadores y sus tipos.
8                   for (String id : idents) {
9                           //chequear unicidad
10                          if (tablaActual.getTipos().containsKey(id) ||
                            ↪  tablaActual.getNombre().equals(id)) {
11                                  errorSemantico("unicidad", id);
12                          } else {
13                                  tablaActual.addIdentificador(id, type);
14                          }
15                  }
16          } else {
17                  errorSintactico("TK_ID");
18          }
19  }
```

Declaración de procedimientos: cuando se encuentra un procedimiento se crea un nuevo ambiente. El ambiente nuevo recibe la clase de ambiente, el nombre del identificador del ambiente, y el ambiente padre. Se

recorren las listas de parámetros para agregarlos a ambos ambientes, en el hijo se requieren los identificadores, y en el padre los tipos.

```
1  private void procedure_heading() {
2          if (preanalisis.getNombre().equals("TK_PROCEDURE")) {
3                  match("TK_PROCEDURE");
4                  String nombre = identifier();
5                  LinkedList<LinkedList<String>> listaParametros = new LinkedList<>();
6                  parameters(listaParametros);
7                  //se agrega el identificador al padre
8                  if (tablaActual.getTipos().containsKey(nombre)) {
9                          errorSemantico("unicidad", nombre);
10                 } else {
11                         tablaActual.addProcedure(nombre);
12                 }
13                 //se crea la nueva tabla para el ambiente actual del procedimiento
14                 Ambiente padre = tablaActual;
15                 tablaActual = new Ambiente("TK_PROCEDURE", nombre, padre);
16                 String id;
17                 String type;
18                 int i = 0;
19                 LinkedList<String> aux;
20                 while (i < listaParametros.size()) {
21                         aux = listaParametros.get(i);
22                         type = aux.get(0);
23                         for (int j = 1; j < aux.size(); j++) {
24                                 id = aux.get(j);
25                                 if (tablaActual.getTipos().containsKey(id) ||
                                   ↪  tablaActual.getNombre().equals(id)) {
26                                         errorSemantico("unicidad", id);
27                                 } else {
28                                         tablaActual.addIdentificador(id, type);
29                                         //se le asigna al padre
30                                         tablaActual.getPadre().addParametro(nombre,
                                           ↪  type);
31                                 }
32                         }
33                         i++;
34                 }
35         } else {
36                 errorSintactico("TK_PROCEDURE");
37         }
38 }
```

Verificación de compatibilidad de tipos: cuando ocurre una operación se debe verificar que sus operandos se correspondan con los tipos definidos en la tabla 5.1. En estos casos, siempre se recibe un atributo heredado que es el tipo del primer operando, y un atributo sintetizado que es el tipo del segundo operando (con excepción de las operaciones unarias).

```
1  private String expression_or_1(String type) {
2          if (preanalisis.getNombre().equals("TK_BOOL_OP_OR")) {
3                  match("TK_BOOL_OP_OR");
4                  String type2 = expression_and();
5                  if (not((type.equalsIgnoreCase(type2)) and
                   ↪  type.equalsIgnoreCase("TK_TYPE_BOOL"))) {
6                          errorSemantico("type", type + " y " + type2 + " no aplicables a
                           ↪  operador OR");
```

```
                }
                type = "TK_TYPE_BOOL";
                type = expression_or_1(type);
        }
        return type;
}
```

Para ver el flujo de datos que ocurren durante el análisis, se muestran a continuación árboles de derivación anotados, donde se aprecia el uso de los ambientes y los atributos de la gramática embebidos en el análisis sintáctico.

Para simplificar el nombre de los nodos, se abrevió el nombre de los no terminales de la gramática, quedando como los de la siguiente lista:

- VD: Variable_declaration
- IL: Identifier_list
- I: Identifier
- IL_1: Identifier_list_1
- VDL: variable_declaration_list
- VDL_1: variable_declaration_list_1
- PD: procedure_declaration
- PH: procedure_heading
- P: parameters
- P_2: parameters_2
- PDL: parameter_declaration_list
- PrD: parameter_declaration

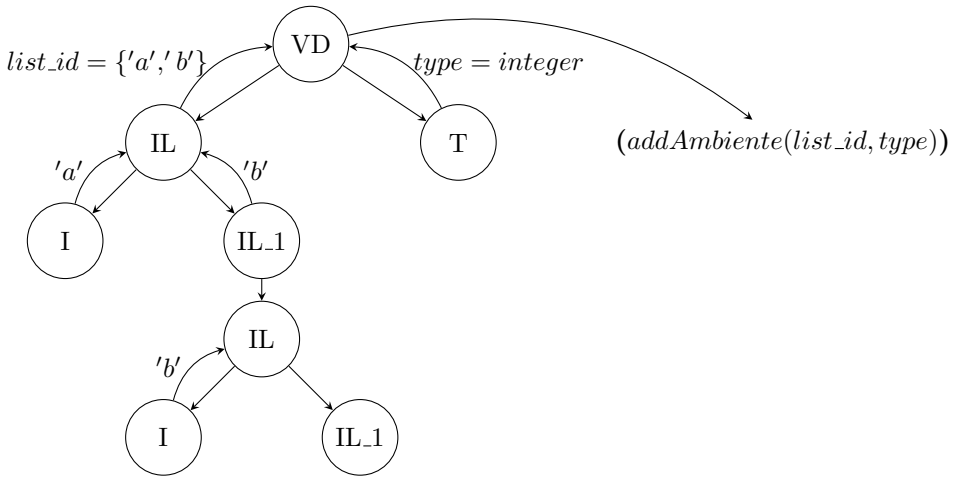


Figura 5.2: Declaración simple de variables.

En la figura 5.2 se ve un árbol de derivación anotado teniendo en cuenta la gramática de atributos y el uso de la tabla de símbolos, para la definición de dos variables integer: “VAR a,b:integer;”. Cuando el no terminal VD obtiene el atributo list_id que tiene la lista de identificadores declarados y type que tiene el tipo de dato de los identificadores, puede insertarlos en el Ambiente actual, que representa la tabla de símbolos de la subrutina que está analizando.

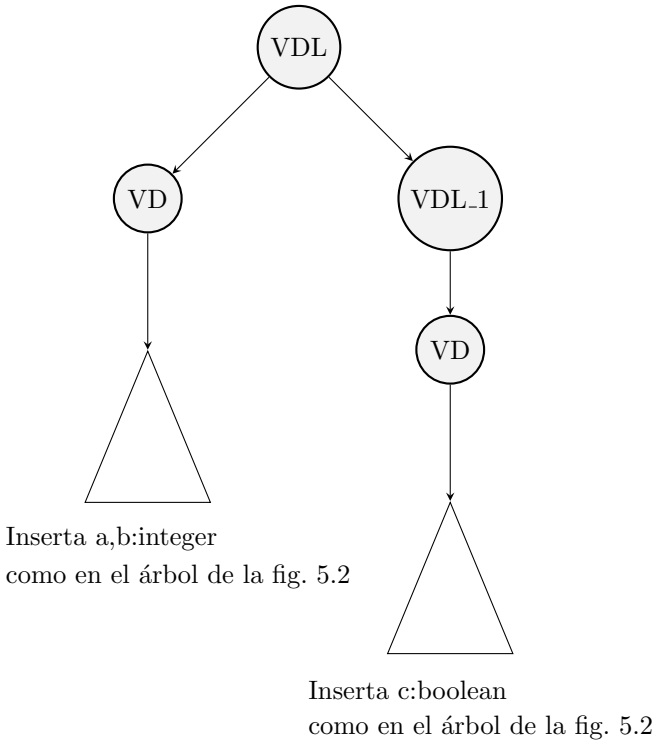


Figura 5.3: Declaración de listas de variables.

En la figura 5.3 se ve el árbol de derivación para insertar una lista de variables con la siguiente sintaxis: “VAR a,b:integer; c:boolean”. Así, la subrutina que está siendo analizada va a recibir todas sus variables sin importar cómo sea la declaración de variables.

En el siguiente árbol de la figura 5.4 veremos la declaración de un procedimiento con parámetros, similar a la declaración de variables pero jugando un poco más con la tabla de símbolos.

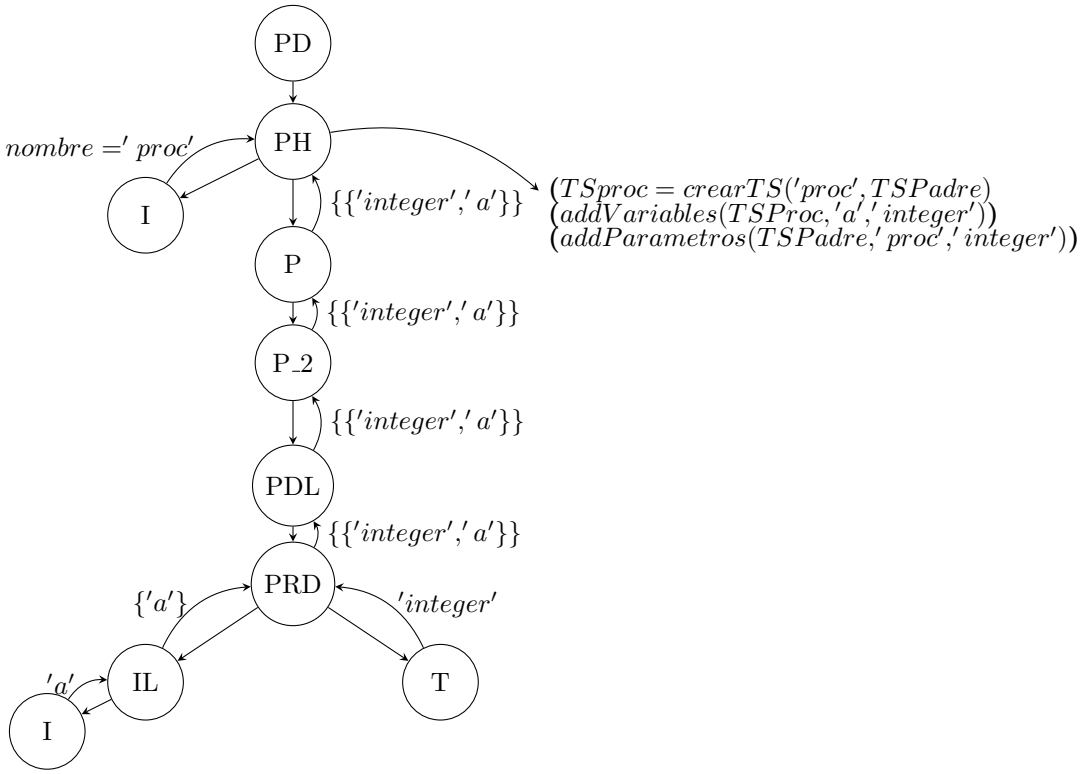


Figura 5.4: Declaración simple de variables.

En la figura 5.4 se muestra el análisis para la declaración del encabezado de un procedimiento con la sintaxis: "procedure proc(a:integer);". El árbol muestra cómo se obtiene el nombre del procedimiento, sus parámetros y luego en el nodo PH se procede a crear la nueva tabla de símbolos para ese procedimiento, la cual recibe como parámetro el nombre del procedimiento y la tabla de símbolos del padre (teniendo en cuenta su estructura lexicográfica). Luego se agregan los parámetros como variables a la tabla del procedimiento; y se agregan los tipos de los parámetros a la tabla del procedimiento padre, para que cuando se invoque a *proc*, pueda chequear la cantidad y el tipo de los parámetros que se le envían.

Si la declaración de parámetros fuera la siguiente: (a,b:integer;c:boolean), en el nodo PRD se armaría la siguiente lista: {{'integer','a','b'},{'boolean','c'}}, y una vez que llegue a PH se ejecutaría addVariables(TSProc,'a','integer'), addVariables(TSProc,'b','integer'), addVariables(TSProc,'c','boolean') y addParametros(TSPadre,'proc',{'integer','integer','boolean'}).

5.6.4. Instructivos de instalación y uso

Para usar el programa es idéntico que para el Análizador Sintáctico, al ser un programa portable, solo se requiere la compilación a través del comando *javac *.java* o cargando el proyecto en el NetBeans y usando este para compilar.

Para ejecutar, hay que invocar el programa compilado enviándole como parámetro el archivo con el código fuente del programa a compilar.

También es posible ejecutar la batería de ejemplos mediante el ejecutable `bateria-wind.bat` para Windows o `bateria-linux.sh` para Linux.

5.6.5. Ejemplos

Para mostrar el funcionamiento del Analizador Semántico proponemos una serie de ejemplos para demostrar las funcionalidades semánticas implementadas.

Chequeo de unicidad

```
1  Program Example1;
2  Var
3      Num1, Num2, Sum1 : Integer;
4      Num1 : Boolean;
5  Begin
6      Sum := Num1 + Num2;
7      if (Num1 > Num2) then
8          Result := true
9  End.
```

Figura 5.5: Programa en Pascal reducido con error de unicidad en variable Num1.

```
ejemplo.pas: error de compilación.
Error semantico: linea 4.
Identificador Num1 ya declarado en el ambiente.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 5.6: Resultado compilación ejemplo1.

Chequeo de existencia

```
1  Program Example2;
2  Var
3      Num1, Num2, Sum1 : Integer;
4  Begin
5      Num1 := 2;
6      Sum2 := Num1 + Num2;
7  End.
```

Figura 5.7: Programa en Pascal reducido con falta de declaración de variable Sum2.

```
ejemplo.pas: error de compilación.
Error semantico linea 6.
Identificador no declarado.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 5.8: Resultado compilación ejemplo2.

Correspondencia de parámetros

```
1  Program Example3;
2  Var
3      Num1, Num2, Sum1 : Integer;
4      Procedure proc(a:integer, b:boolean);
5      begin
6          b := true;
7      end;
8  Begin
9      proc(Num1,true);
10     proc(Num1,Num2)
11 End.
```

Figura 5.9: Programa en Pascal reducido con error en la llamada al procedimiento proc en la línea 10.

```
ejemplo.pas: error de compilación.
Error semantico linea 10.
La lista de parametros no coincide con la definicion de la subrutina
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 5.10: Resultado compilación ejemplo3.

Devolver valor de una función

```
1  Program Example4;
2  Var
3      Num1, Num2, Sum1 : integer;
4      function f1(a:integer): integer ;
5      begin
6          f1 := a+3;
7      end;
8  Begin
9      f1(Num1)
10 End.
```

Figura 5.11: Programa en Pascal reducido con uso de identificador f1 para devolver un valor.

```
Asignacion de retorno dentro de una funcion en la linea 6.
ejemplo.pas: compilación exitosa.
BUILD SUCCESSFUL (total time: 1 second)
```

Figura 5.12: Resultado compilación ejemplo4.

Compatibilidad de tipos

```
1  Program Example5;
2  Var
3      Num1, Num2, Sum1 : Integer;
4      Procedure proc(a:integer, b:boolean);
5      begin
6          a := 3;
7          b := a;
8      end;
9  Begin
10     proc(Num1,Num2)
11 End.
```

Figura 5.13: Programa en Pascal reducido con error en la compatibilidad de tipos línea 7.

```
ejemplo.pas: error de compilación.
Error semantico linea 7.
Se esperaba un TK_TYPE_BOOL pero se encontró TK_TYPE_INT
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 5.14: Resultado compilación ejemplo5.

5.7. Conclusión

Cumplimos con los requerimientos propuestos en la descripción del problema 5.2, extendiendo el Analizador Sintáctico con las reglas semánticas y utilizando los atributos descritos en 5.5.2.

Capítulo 6

Generación de código intermedio

6.1. Introducción

Luego de realizar todos los chequeos sintácticos y semánticos propuestos, llega la etapa de generar código ejecutable.

En este capítulo desarrollaremos la especificación y la implementación de la generación de código intermedio del lenguaje, utilizando código de tres direcciones, y generando un código ejecutable por una máquina virtual.

6.2. Descripción del problema

En esta etapa necesitamos traducir las sentencias del lenguaje pascal a instrucciones de bajo nivel. Para esto, nuestro lenguaje intermedio será MEPa, cuyas instrucciones son ejecutables por la máquina virtual MEPa.

El objetivo es ampliar la funcionalidad del analizador semántico, embebiéndole sentencias para la generación de instrucciones MEPa.

El resultado de la compilación debe ser un archivo con el código MEPa que resulte equivalente al código en lenguaje Pascal y permita la ejecución del mismo.

6.3. Máquina virtual MEPa

MEPa es una máquina hipótetica desarrollada teniendo en cuenta las construcciones de Pascal. Nos permite generar código ejecutable sin necesidad de tener en cuenta las particularidades y complejidad de una máquina real.

La memoria de MEPa se divide en tres secciones: la región del programa, donde almacenará el código del programa; un vector de direcciones base (display) que tendrá punteros a direcciones de la tercer región, la pila, donde almacenará los datos que serán utilizados por las operaciones.

El uso de una pila nos permite implementar fácilmente la evaluación de expresiones, ya que la precedencia de operadores queda implícita por el recorrido del árbol de derivación que surge de nuestra sintaxis. Además, la implementación de las llamadas recursivas también se beneficia al utilizar una estructura de pila para almacenar los valores y las direcciones de retorno.

En la tabla 6.1 se ven las instrucciones que utilizamos y su funcionalidad.

6.4. Estrategias

Para facilitar el trabajo con los saltos condicionales de MEPa, almacenamos los *labels* de cada ambiente en la tabla de símbolos.

Como no tuvimos que preocuparnos por el tamaño de los tipos de datos, para calcular los offsets de las variables utilizamos su posición en el orden de declaración, y se guardó esta posición para cada variable en la tabla de símbolos. Para calcular la memoria a reservar, contamos la cantidad de variables que había en el ambiente.

	Instrucción	Descripción
Operaciones aritméticas	SUMA	sumar
	SUST	restar
	MULT	multiplicar
	DIVI	dividir
Operaciones relacionales	CMIG	igual
	CMDG	desigual
	CMME	menor
	CMNI	menor igual
	CMMA	mayor
	CMYI	mayor igual
Operaciones unarias	NEGA	negación lógica
	UMEN	menos unario
Operaciones lógicas	CONJ	and
	DISJ	or

	Instrucción	Descripción
Salto condicional	DSVS label	desviar siempre
	DSVF label	desviar si el tope es falso
Apilar	APCT c	apilar constante
	APVL m,n	apilar variable
Almacenar	ALVL m,n	almacena el tope de la pila en una variable
Entrada/Salida	IMPR	imprimir por pantalla
	LEER	leer entrada
Programas y procedimientos	INPP	inicio del programa
	LLPR label	llamada a subrutina
	ENPR k	entrada a procedimiento
	RTPR k,n	retornar desde un procedimiento
	RMEM c	reservar memoria
	LMEM c	liberar memoria
	PARA	detiene la máquina MEPa
	NADA	sin efecto en la ejecución

Tabla 6.1: Instrucciones MEPa

6.5. Limitaciones

Algunas operaciones de MEPa no fueron utilizadas, como salidas por pantalla o lecturas de teclado que generen saltos de línea automáticos; aunque sean más una comodidad para el programador que una limitación.

6.6. Diseño del generador de código intermedio

Para especificar cómo se llevará a cabo la generación de código intermedio, se describirán las modificaciones necesarias sobre las reglas de la gramática.

Para cumplir los objetivos de 6.2, se utilizará una variable a la cuál se le agregará el código MEPa que se genere durante la compilación. Una vez concluida la compilación el contenido de la variable se escribirá en un archivo.

6.6.1. Modificaciones a la tabla de símbolos

La tabla de símbolos será actualizada con nuevos valores necesarios para la generación de código intermedio. Estos valores están asociados a la profundidad y desplazamiento de los identificadores, necesarios para implementar el alcance estático a través del vector de direcciones y la pila de MEPa.

6.6.2. Modificaciones a las reglas de la gramática

Cada regla que esté asociada a la generación de instrucciones de la tabla 6.1, será modificada, agregándole a la variable que contiene el código MEPa la instrucción correspondiente.

- <program> agrega `INPP` antes de llamar a <program-heading> y luego de unificar con el "." final agrega `PARA`. Esto indica el comienzo y fin del programa principal.
- En <block>, luego de llamar <declaration-block> y <multiple-statement> agrega `LMEM X`. Esto libera la memoria asignada a las variables, donde `X` es la cantidad de variables que se reservaron en <declaration-block>.
- <declaration-block> agrega `RMEM X` luego de <variable-declaration-block>, reservando `X` posiciones de memoria para las variables locales.
- <procedure-declaration> y <function-declaration>, agrega `DSVS` $\texttt{LX}_1$ la cuál es una instrucción de salto, y donde $\texttt{X}_1$ es un label que corresponde con el final del subprograma. Luego agrega $\texttt{LX}_2$ `ENPR P` que

indica el comienzo del subprograma, y donde X_2 es un label que se utilizará para llamar al subprograma posteriormente y P es la profundidad del ambiente actual. Finalmente se agrega RTPR P X_3 y LX_1 NADA, para indicar el retorno del subprograma, donde P es la misma profundidad mencionada anteriormente, X_3 es la cantidad de parámetros recibidos por el subprograma, y por último X_1 es el label definido para el final del subprograma y donde se desviará la primera vez, durante la definición.

- <assignment-statement> agrega ALVL P O que permite almacenar el valor del tope de la pila en una variable indicada mediante la profundidad P y el offset O.
- <call-procedure-or-function> agrega varias instrucciones diferentes dependiendo del tipo de subprograma llamado:
  - Si es un procedimiento *write* agrega IMPR que permite imprimir por pantalla el contenido del tope de la pila.
  - Si es un procedimiento *read* agrega LEER que activa la entrada por teclado y luego realiza ALVL P O para almacenar el valor leído en la variable indicada por la profundidad P y el offset O.
  - Si es un procedimiento definido por el usuario, agrega LLPR LX que indica la llamada al procedimiento que fue etiquetado con el label X.
  - Si es una función definida por el usuario, agrega LLPR LX que indica la llamada a la función que fue etiquetado con el label X y además agrega RMEM 1 para reservar una posición de memoria extra para guardar el valor de retorno de la función.
- <conditional-statement> agrega DSVF LX_1 que permite saltar el código hasta la instrucción etiquetada con X_1 si el tope de la pila es falso, es decir no ejecuta el cuerpo del *then* en caso de ser falsa la condición de if.
- <else-statement> agrega DSVS LX_2 para desviar el cuerpo del *else* en caso de que el contenido del *then* se haya ejecutado. Luego del desvió agrega LX_1 NADA que será donde se desviará en caso de que el *then* sea falso, y permitirá la ejecución del *else*, ya que este no será desviado. Finalmente agrega LX_2 NADA que será donde se desviará el *else* en caso de no cumplirse. Si no se detecto la sentencia *else* por el analizador sintáctico solo se agrega LX_1 NADA que corresponde a donde se desviará el *then* en caso de ser falso.
- <repetitive-statement> agrega LX_1 NADA que será la instrucción etiquetada donde se desviará el flujo en caso de que la condición de repetición se cumpla. Luego agrega DSVF LX_2 y DSVS LX_1 indicando que se desviará a la instrucción etiquetada X_2 en caso de no cumplirse la condición del *while* o se desviará a X_1 en caso de cumplirse. Finalmente se agrega LX_2 NADA que es la instrucción a la que se desviará el flujo en caso de que no se cumpla la condición de repetición.
- <expression-or-1> agrega DISJ que indica la disyunción entre los últimos dos valores booleanos almacenados en el tope de la pila.
- <expression-and-1> agrega CONJ que indica la conjunción entre los últimos dos valores booleanos almacenados en el tope de la pila.
- <expression-rel-1> agrega varias instrucciones diferentes dependiendo de la operación relacional que se realiza utilizando los últimos dos valores booleanos o enteros almacenados en el tope de la pila:
  - Agrega CMIG para la relación igual.
  - Agrega CMDG para la relación desigual.
  - Agrega CMME para la relación menor.
  - Agrega CMMA para la relación mayor.
  - Agrega CMNI para la relación menor igual.
  - Agrega CMYI para la relación mayor igual.
- <expression-add-1> agrega SUMA o SUST para el caso de la operación suma o resta, según corresponda, entre los últimos dos valores enteros almacenados en el tope de la pila.

- <expression-mult-1> agrega `MULT` o `DIVI` para el caso de la operación multiplicación o división, según corresponda, entre los últimos dos valores enteros almacenados en el tope de la pila.
- <factor> agrega `UMEN` o `NEGA` para el caso de la operación unaria negación aritmética o negación relacional, según corresponda, entre los últimos dos valores enteros o booleanos almacenados en el tope de la pila. En caso de que el factor sea un identificador de variable o parámetro agrega `APVL P O` con la profundidad `P` y el offset `O` correspondiente a tal variable o parámetro para almacenar tal valor en el tope de la pila.
- <number> agrega `APCT C` que almacena una constante numérica `C` en el tope de la pila.
- <bool> agrega `APCT 1` o `APCT 0` según sea un valor constante *true* o $false$ correspondientemente, lo cuál almacena tal valor en el tope de la pila.

6.7. Implementación del aplicativo

6.7.1. Descripción del problema

En esta implementación debemos generar el código intermedio según las sentencias del archivo de entrada en lenguaje Pascal.

En esta etapa no existen errores de generación de código, ya que simplemente es devolver una secuencia de instrucciones, por lo que los únicos errores que lanzará el compilador son los de las etapas previas de análisis del código fuente. Si existe un error en alguna de las etapas de análisis, no se geenerará ningún archivo de salida con código MEPa.

La implementación se basará en agregar las sentencias necesarias para crear el archivo de salida con el código MEPa. Estas sentencias estarán embebidas en el analizador semántico.

6.7.2. Herramientas utilizadas

Continuamos utilizando el mismo proyecto del analizador semántico, en lenguaje Java con la misma versión y Sistema Operativo Windows 10.

Para verificar el funcionamiento del código MEPa generado, utilizamos el aplicativo que se encuentra en la plataforma PEDCo.

6.7.3. Diseño

Como la generación de código es simplemente agregar las instrucciones MEPa a una variable, y luego retornar la variable, trabajaremos sobre los procedimientos implementados en el Analizador Semántico.

Para no perder la implementación del Analizador Semántico, crearemos otra clase llamada **GeneradorCodigoIntermedio** dentro del paquete **codigointermedio** con el mismo código del **AnalizadorSemántico**, y además crearemos otro TDA **Ambiente** modificado, por lo que nuestro árbol de directorios quedará como en la figura 6.1.

```
src
└── compiladorpascal
    ├── CompiladorPascal.java
    ├── lexico
    ├── sintactico
    ├── semantico
    └── codigointermedio
        ├── GeneradorCodigoIntermedio.java
        └── Ambiente.java
```

Figura 6.1: Árbol de directorios del proyecto de Java con los archivos del generador de código intermedio.

La tabla de símbolos requirió agregar información tanto para poder representar el alcance estático, como para identificar qué identificadores dentro de un ambiente son variables y cuáles son parámetros, para poder calcular el correcto acceso de los valores de la pila. A continuación se puedne ver las modificaciones sobre el TDA Ambiente, donde se aprecia la información necesaria descrita en 6.6.1.

```
public class Ambiente {

    ...

    //profundidad del ambiente
    private int profundidad = -1;
    //ultimo offset utilizado para una variable
    private int lastOffset = 0;
    //ultimo offset utilizado para un parametro
    private int lastParameterOffset = -3;
    //offset utilizado para el retorno de una funcion
    private int returnOffset = -4;
    //label de llamada para un procediemiento o funcion
    private int label = -1;

    //Asocia un identificador (variable) a su profundidad
    private HashMap<String, Integer> profundidades;
    //Asocia un identificador (variable) a su offset
    private HashMap<String, Integer> offsets;
    //Asocia un identificador a su clase (variable, parametro, funcion, procedimiento)
    private HashMap<String, String> clases;
    //Asocia un identificador de procedimiento o funcion a su label de llamada
    private HashMap<String, Integer> labels;

    ...

    //Asocia un identificador a su type y profundidad.
    public void addVariable(String id, String tipo, int profundidad, boolean
    ↪  isParameter) {
        tipos.put(id.toUpperCase(), tipo.toUpperCase());
        profundidades.put(id.toUpperCase(), profundidad);
        if (isParameter) {
            offsets.put(id.toUpperCase(), lastParameterOffset++);
            clases.put(id.toUpperCase(), "parametro");
        } else {
            offsets.put(id.toUpperCase(), lastOffset++);
            clases.put(id.toUpperCase(), "variable");
        }
    }
    ...
}
```

Como los parámetros son enviados desde el llamador, la subrutina invocada debe acceder a ellos con un offset negativo (desde el comienzo de su registro de activación accede a posiciones anteriores de la pila), mientras que para acceder a las variables locales, debe acceder con un offset positivo (dentro de su espacio local de la pila). Por esto, al agregar esta información en la tabla de símbolos, necesitamos discriminar la clase del identificador. Esta funcionalidad se agregó al método *addVariable()* utilizando un parámetro más que indica si la variable insertada es un parámetro.

Para mostrar el uso de la nueva información de la tabla de símbolos se expondrán fragmentos de los procedimientos más importantes.

```
 private String factor() {
    ...
    if (ambiente.getClase(id).equals("variable") ||
    ↪  ambiente.getClase(id).equals("parametro")) {
        mepa += "APVL " + ambiente.getProfundidad(id) + " " + ambiente.getOffset(id) +
        ↪  "\n";
```

```
5           } else {
6               if (ambiente.getClase(id).equalsIgnoreCase("funcion")) {
7                   call_procedure_or_function(id);
8               } else {
9               errorSemantico("bad_uso_proc", "No se permite uso de procedimientos en
                ↪  expresiones. Causa: retorno void.");
10              }
11      ...
12  }
```

En el procedimiento *factor()* se puede apreciar el uso de la clase de un identificador para verificar si es una variable o un parámetro, y cómo se accede a su profundidad y offset para almacenar su valor en la pila con la instrucción APVL.

6.7.4. Instructivos de instalación y uso

La instalación y uso es idéntica que para el analizador semántico. Se puede compilar a través del comando *javac *.java* o cargando el proyecto en NetBeans y utilizando este para compilar.

Para ejecutar, se debe invocar el programa compilado enviándole como parámetro el archivo con el código fuente del programa a compilar.

Como resultado de una compilación exitosa, se debe obtener un archivo de salida con el código MEPa listo para ejecutar en la máquina virtual.

6.7.5. Ejemplos

Asignación y expresión simple

```
1  Program MepaExpresion;
2  Var
3      a, b, c : integer;
4  Begin
5      a := a + (b / 9 - 3) * c
6  End.
```

Figura 6.2: Programa en Pascal reducido ilustrando una asignación de una expresión simple.

```
1   INPP
2   RMEM 3
3   APVL 0 0
4   APVL 0 1
5   APCT 9
6   DIVI
7   APCT 3
8   SUST
9   APVL 0 2
10  MULT
11  SUMA
12  ALVL 0 0
13  LMEM 3
14  PARA
```

Figura 6.3: Salida de la ejecución del generador de código intermedio con el código de la figura 6.2.

Variables locales de un subprograma

```
1  Program MepaProcedimiento;
2  Var
3      k:integer;
4  procedure p (n:integer; g:integer);
5  Var
6      h:integer;
7  Begin
8      if n<2 then h:=g+n
9      else begin
10         h:=g;
11         p(n-1,h);
12         k:=h;
13         p(n-2,g)
14     end;
15     write(n)
16 End;
17 Begin
18     k:=0;
19     p(3,k)
20 End.
```

Figura 6.4: Programa en Pascal reducido ilustrando la definición y llamada de un procedimiento.

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

Figura 6.5: Salida de la ejecución del generador de código intermedio con el código de la figura 6.4.

Pasaje de parámetros para una función

El pasaje de parámetros para una función es similar al de un procedimiento, con la diferencias que se reserva un lugar de memoria más en la pila antes de efectuar la llamada, para obtener el resultado de la función. Esta posición de memoria extra puede verse como un parámetro adicional, que no se desapila luego del retorno de la función.

```
Program MepaFuncion;
Var
    a: integer;
function fact(n:integer):integer;
Begin
    if n < 2 then fact := 1
    else fact := fact(n-1)*n
End;
Begin
    a:=3;
    write (fact(a))
End.
```

Figura 6.6: Programa en Pascal reducido ilustrando la definición y llamada de una función.

```
INPP
RMEM 1
DSVS L1
L2 ENPR 1
APVL 1 -3
APCT 2
CMME
DSVF L3
APCT 1
ALVL 1 -4
DSVS L4
L3 NADA
RMEM 1
APVL 1 -3
APCT 1
SUST
LLPR L2
APVL 1 -3
MULT
ALVL 1 -4
L4 NADA
RTPR 1 1
L1 NADA
APCT 3
ALVL 0 0
RMEM 1
APVL 0 0
LLPR L2
IMPR
LMEM 1
PARA
```

Figura 6.7: Salida de la ejecución del generador de código intermedio con el código de la figura 6.6.

Sentencia condicional

```
Program MepaCondicional;
Var
    a, b : integer;
    p, q: boolean;
Begin
    if a > b then q := p and q
        else if (a < 2*b) then p := true
            else q := false
End.
```

Figura 6.8: Programa en Pascal reducido ilustrando una sentencia condicional.

```
1  INPP
2  RMEM 4
3  APVL 0 0
4  APVL 0 1
5  CMMA
6  DSVF L1
7  APVL 0 2
8  APVL 0 3
9  CONJ
10 ALVL 0 3
11 DSVS L2
12 L1 NADA
13 APVL 0 0
14 APCT 2
15 APVL 0 1
16 MULT
17 CMME
18 DSVF L3
19 APCT 1
20 ALVL 0 2
21 DSVS L4
22 L3 NADA
23 APCT 0
24 ALVL 0 3
25 L4 NADA
26 L2 NADA
27 LMEM 4
28 PARA
```

Figura 6.9: Salida de la ejecución del generador de código intermedio con el código de la figura 6.8.

Sentencia repetitiva

```
1 Program MepaRepetitiva;
2 Var
3     s, n : integer;
4 Begin
5     while s <= n do
6         s := s + 3
7 End.
```

Figura 6.10: Programa en Pascal reducido ilustrando una sentencia repetitiva.

```
1  INPP
2  RMEM 2
3  L1 NADA
4  APVL 0 0
5  APVL 0 1
6  CMNI
7  DSVF L2
8  APVL 0 0
9  APCT 3
10 SUMA
11 ALVL 0 0
12 DSVS L1
13 L2 NADA
14 LMEM 2
15 PARA
```

Figura 6.11: Salida de la ejecución del generador de código intermedio con el código de la figura 6.10.

6.8. Problemas encontrados

Durante la generación de funciones, nos encontramos que si una función era llamada y su valor de retorno no era usado, la pila de MEPa quedaba con memoria reservada, lo que generaba problemas cuando se hacían retornos al procedimiento llamador. Por esto, evitamos el uso de funciones cuyo valor de retorno no sea usado.

6.9. Posibles mejoras

Puede realizarse una optimización de las instrucciones generadas, sobre todo si se encuentra algún flujo del programa que nunca se ejecute, por ejemplo en una sentencia condicional con la forma if (false) {instrucciones...}.

6.10. Conclusión

Pudimos generar el código intermedio MEPa cumpliendo todas las características propuestas. El conjunto de instrucciones resultante no está optimizado, por lo que pueden haber instrucciones innecesarias que reduzcan

la eficiencia del programa.

Como consecuencia de trabajar sobre el mismo código que el Analizador Semántico, el programa creció y se dificulta su lectura.

Apéndice

A.1. Pseudocódigo Analizador Sintáctico

```
void match(terminal) {
 if (preanalisis == terminal) {
  preanalisis = tokenSiguiente();
 } else {
  error();
 }
}

void program() {
 if (preanalisis == TK_PROGRAM) {
  program_heading();
  block();
  match(TK_POINT);
 } else {
  error();
 }
}

void program_heading() {
 if (preanalisis == TK_PROGRAM) {
  match(TK_PROGRAM);
  identifier();
  match(TK_ENDSTNC);
 } else {
  error();
 }
}

void block() {
 switch (preanalisis) {
  case TK_VAR:
  case TK_PROCEDURE:
  case TK_FUNCTION:
   declaration_block();
   multiple_statement();
   break;
  case TK_BEGIN:
   multiple_statement();
   break;
  default:
   error();
   break;
 }
}
```

```
void declaration_block() {
 switch (preanalisis) {
  case TK_VAR:
   variable_declaration_block();
   declaration_block_1();
   break;
  case TK_PROCEDURE:
  case TK_FUNCTION:
   declaration_block_1();
   break;
  default:
   error();
   break;
 }
}

void declaration_block_1() {
 switch (preanalisis) {
  case TK_PROCEDURE:
  case TK_FUNCTION:
   procedure_and_function_declaration_list();
   break;
 }
}

void variable_declaration_block() {
 if (preanalisis == TK_VAR)) {
  match(TK_VAR);
  variable_declaration_list();
 } else {
  error();
 }
}

void variable_declaration_list() {
 if (preanalisis == TK_ID)) {
  variable_declaration();
  match(TK_ENDSTNC);
  variable_declaration_list_1();
 } else {
  error();
 }
}
```

```
void variable_declaration_list_1() {
 if (preanalisis == TK_ID)) {
  variable_declaration_list();
 }
}

void variable_declaration() {
 if (preanalisis == TK_ID)) {
  identifier_list();
  match(TK_TPOINTS);
  type();
 } else {
  error();
 }
}

void procedure_and_function_declaration_list() {
 switch (preanalisis) {
  case TK_PROCEDURE:
   procedure_declaration();
   match(TK_ENDSTNC);
   procedure_and_function_declaration_list_1();
   break;
  case TK_FUNCTION:
   function_declaration();
   match(TK_ENDSTNC);
   procedure_and_function_declaration_list_1();
   break;
  default:
   error();
   break;
 }
}

void procedure_and_function_declaration_list_1() {
 switch (preanalisis) {
  case TK_PROCEDURE:
  case TK_FUNCTION:
   procedure_and_function_declaration_list();
   break;
 }
}

void procedure_declaration() {
 if (preanalisis == TK_PROCEDURE)) {
  procedure_heading();
  match(TK_ENDSTNC);
  block();
 } else {
  error();
 }
}

void procedure_heading() {
 if (preanalisis == TK_PROCEDURE)) {
  match(TK_PROCEDURE);
  identifier();
  parameters();
 } else {
  error();
 }
}

void function_declaration() {
 if (preanalisis == TK_FUNCTION)) {
  function_heading();
  match(TK_ENDSTNC);
  block();
 } else {
  error();
 }
}

void function_heading() {
 if (preanalisis == TK_FUNCTION)) {
  match(TK_FUNCTION);
  identifier();
  parameters();
  match(TK_TPOINTS);
  type();
 } else {
  error();
 }
}

void parameters() {
 if (preanalisis == TK_OPAR)) {
  match(TK_OPAR);
  parameters_2();
  match(TK_CPAR);
 }
}

void parameters_2() {
 if (preanalisis == TK_ID)) {
  parameter_declaration_list();
 }
}

void parameter_declaration_list() {
 if (preanalisis == TK_ID)) {
  parameter_declaration();
  parameter_declaration_list_1();
 } else {
  error();
 }
}

void parameter_declaration_list_1() {
 if (preanalisis == TK_COMMA)) {
  match(TK_COMMA);
  parameter_declaration_list();
 }
}

void parameter_declaration() {
 if (preanalisis == TK_ID)) {
  identifier_list();
  match(TK_TPOINTS);
  type();
 } else {
  error();
 }
}
```

```
void statement_block() {
 switch (preanalisis) {
  case TK_ID:
  case TK_WRITE:
  case TK_READ:
  case TK_IF:
  case TK_WHILE:
   statement();
   break;
  case TK_BEGIN:
   multiple_statement();
   break;
 }
}

void multiple_statement() {
 if (preanalisis == TK_BEGIN)) {
  match(TK_BEGIN);
  statement_list();
  match(TK_END);
 } else {
  error();
 }
}

void statement_list() {
 switch (preanalisis) {
  case TK_ID:
  case TK_WRITE:
  case TK_READ:
  case TK_IF:
  case TK_WHILE:
   statement();
   statement_list_1();
   break;
  default:
   error();
   break;
 }
}

void statement_list_1() {
 if (preanalisis == TK_ENDSTNC)) {
  match(TK_ENDSTNC);
  statement_list();
 }
}
```

```
void statement() {
 switch (preanalisis) {
  case TK_ID:
  case TK_WRITE:
  case TK_READ:
   simple_statement();
   break;
  case TK_IF:
  case TK_WHILE:
   structured_statement();
   break;
  default:
   error();
   break;
 }
}

void simple_statement() {
 switch (preanalisis) {
  case TK_ID:
   identifier();
   simple_statement_1();
   break;
  case TK_WRITE:
   match(TK_WRITE);
   call_procedure_or_function();
   break;
  case TK_READ:
   match(TK_READ);
   call_procedure_or_function();
   break;
  default:
   error();
   break;
 }
}

void simple_statement_1() {
 switch (preanalisis) {
  case TK_ASSIGN:
   assignment_statement();
   break;
  case TK_OPAR:
   call_procedure_or_function();
   break;
 }
}

void structured_statement() {
 switch (preanalisis) {
  case TK_IF:
   conditional_statement();
   break;
  case TK_WHILE:
   repetitive_statement();
   break;
  default:
   error();
   break;
 }
}
```

```
void assignment_statement() {
 if (preanalisis == TK_ASSIGN)) {
  match(TK_ASSIGN);
  expression_or();
 } else {
  error();
 }
}

void call_procedure_or_function() {
 if (preanalisis == TK_OPAR)) {
  match(TK_OPAR);
  call_procedure_or_function_1();
  match(TK_CPAR);
 } else {
  error();
 }
}

void call_procedure_or_function_1() {
 switch (preanalisis) {
  case TK_ID:
  case TK_OPAR:
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   expression_list();
   break;
 }
}

void conditional_statement() {
 if (preanalisis == TK_IF)) {
  match(TK_IF);
  expression_or();
  match(TK_THEN);
  statement_block();
  else_statement();
 } else {
  error();
 }
}

void else_statement() {
 if (preanalisis == TK_ELSE)) {
  match(TK_ELSE);
  statement_block();
 }
}

void repetitive_statement() {
 if (preanalisis == TK_WHILE)) {
  match(TK_WHILE);
  expression_or();
  match(TK_DO);
  statement_block();
 } else {
  error();
 }
}
```

```
void expression_list() {
 switch (preanalisis) {
  case TK_ID:
  case TK_OPAR:
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   expression_or();
   expression_list_1();
   break;
  default:
   error();
   break;
 }
}

void expression_list_1() {
 if (preanalisis == TK_COMMA)) {
  match(TK_COMMA);
  expression_list();
 }
}

void expression_or() {
 switch (preanalisis) {
  case TK_ID:
  case TK_OPAR:
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   expression_and();
   expression_or_1();
   break;
  default:
   error();
   break;
 }
}

void expression_or_1() {
 if (preanalisis == TK_OR)) {
  match(TK_OR);
  expression_and();
  expression_or_1();
 }
}
```

```
void expression_and() {
 switch (preanalisis) {
  case TK_ID:
  case TK_OPAR:
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   expression_rel();
   expression_and_1();
   break;
  default:
   error();
   break;
 }
}

void expression_and_1() {
 if (preanalisis == TK_AND)) {
  match(TK_AND);
  expression_rel();
  expression_and_1();
 }
}

void expression_rel() {
 switch (preanalisis) {
  case TK_ID:
  case TK_OPAR:
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   expression_add();
   expression_rel_1();
   break;
  default:
   error();
   break;
 }
}

void expression_rel_1() {
 switch (preanalisis) {
  case TK_REL_OP_EQ:
  case TK_REL_OP_NEQ:
  case TK_REL_OP_MIN:
  case TK_REL_OP_MAY:
  case TK_REL_OP_LEQ:
  case TK_REL_OP_GEQ:
   relational_operator();
   expression_add();
   expression_rel_1();
   break;
 }
}
```

```
void expression_add() {
 switch (preanalisis) {
  case TK_ID:
  case TK_OPAR:
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   expression_mult();
   expression_add_1();
   break;
  default:
   error();
   break;
 }
}

void expression_add_1() {
 switch (preanalisis) {
  case TK_ADD_OP_SUM:
  case TK_ADD_OP_REST:
   addition_operator();
   expression_mult();
   expression_add_1();
   break;
 }
}

void expression_mult() {
 switch (preanalisis) {
  case TK_ID:
  case TK_OPAR:
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   factor();
   expression_mult_1();
   break;
  default:
   error();
   break;
 }
}

void expression_mult_1() {
 switch (preanalisis) {
  case TK_MULT_OP_POR:
  case TK_MULT_OP_DIV:
   multiplication_operator();
   factor();
   expression_mult_1();
   break;
 }
}
```

```
void factor() {
 switch (preanalisis) {
  case TK_ID:
   identifier();
   factor_1();
   break;
  case TK_OPAR:
   match(TK_OPAR);
   expression_or();
   match(TK_CPAR);
   break;
  case TK_ADD_OP_REST:
  case TK_NOT_OP:
   unary_operator();
   factor();
   break;
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
  case TK_NUMBER:
   literal();
   break;
  default:
   error();
   break;
 }
}

void factor_1() {
 if (preanalisis == TK_OPAR)) {
  call_procedure_or_function();
 }
}

void relational_operator() {
 switch (preanalisis) {
  case TK_REL_OP_EQ:
   match(TK_REL_OP_EQ);
   break;
  case TK_REL_OP_NEQ:
   match(TK_REL_OP_NEQ);
   break;
  case TK_REL_OP_MIN:
   match(TK_REL_OP_MIN);
   break;
  case TK_REL_OP_MAY:
   match(TK_REL_OP_MAY);
   break;
  case TK_REL_OP_LEQ:
   match(TK_REL_OP_LEQ);
   break;
  case TK_REL_OP_GEQ:
   match(TK_REL_OP_GEQ);
   break;
  default:
   error();
   break;
 }
}
```

```
void unary_operator() {
 switch (preanalisis) {
  case TK_ADD_OP_REST:
   match(TK_ADD_OP_REST);
   break;
  case TK_NOT_OP:
   match(TK_NOT_OP);
   break;
  default:
   error();
   break;
 }
}

void addition_operator() {
 switch (preanalisis) {
  case TK_ADD_OP_SUM:
   match(TK_ADD_OP_SUM);
   break;
  case TK_ADD_OP_REST:
   match(TK_ADD_OP_REST);
   break;
  default:
   error();
   break;
 }
}

void multiplication_operator() {
 switch (preanalisis) {
  case TK_MULT_OP_POR:
   match(TK_MULT_OP_POR);
   break;
  case TK_MULT_OP_DIV:
   match(TK_MULT_OP_DIV);
   break;
  default:
   error();
   break;
 }
}

void type() {
 switch (preanalisis) {
  case TK_TYPE_INT:
   match(TK_TYPE_INT);
   break;
  case TK_TYPE_BOOL:
   match(TK_TYPE_BOOL);
   break;
  default:
   error(un tipo de dato);
   break;
 }
}
```

```
void identifier_list() {
 if (preanalisis == TK_ID)) {
  identifier();
  identifier_list_1();
 } else {
  error();
 }
}

void identifier_list_1() {
 if (preanalisis == TK_COMMA)) {
  match(TK_COMMA);
  identifier_list();
 }
}

void identifier() {
 if (preanalisis == TK_ID)) {
  match(TK_ID);
 } else {
  error();
 }
}
```

```
void literal() {
 switch (preanalisis) {
  case TK_BOOLEAN_TRUE:
  case TK_BOOLEAN_FALSE:
   bool();
   break;
  case TK_NUMBER:
   number();
   break;
  default:
   error();
   break;
 }
}

void number() {
 if (preanalisis == TK_NUMBER)) {
  match(TK_NUMBER);
 } else {
  error();
 }
}

void bool() {
 switch (preanalisis) {
  case TK_BOOLEAN_TRUE:
   match(TK_BOOLEAN_TRUE);
   break;
  case TK_BOOLEAN_FALSE:
   match(TK_BOOLEAN_FALSE);
   break;
  default:
   error();
   break;
 }
}
```