Compiladores e Intérpretes 2025
Análisis Léxico y Análisis Sintáctico

Grupo 9
Iñaki Roumec : iroumec@alumnos.exa.unicen.edu.ar
Ulises Velis: uvelis@alumnos.exa.unicen.edu.ar

Profesor Asignado
Mailén Gonzalez: mgonzalez@intia.exa.unicen.edu.ar

Introducción

Como trabajo práctico de la cursada de la materia Compiladores e Intérpretes, se busca realizar un compilador, programando y desarrollando a fondo cada una de las siguientes etapas:

- Análisis léxico
- Análisis sintáctico
- Análisis semántico
- Optimizaciones
- Generación de código intermedio
- Generación de código de salida

Basadas en líneas generales en el siguiente gráfico proporcionado por la cátedra.

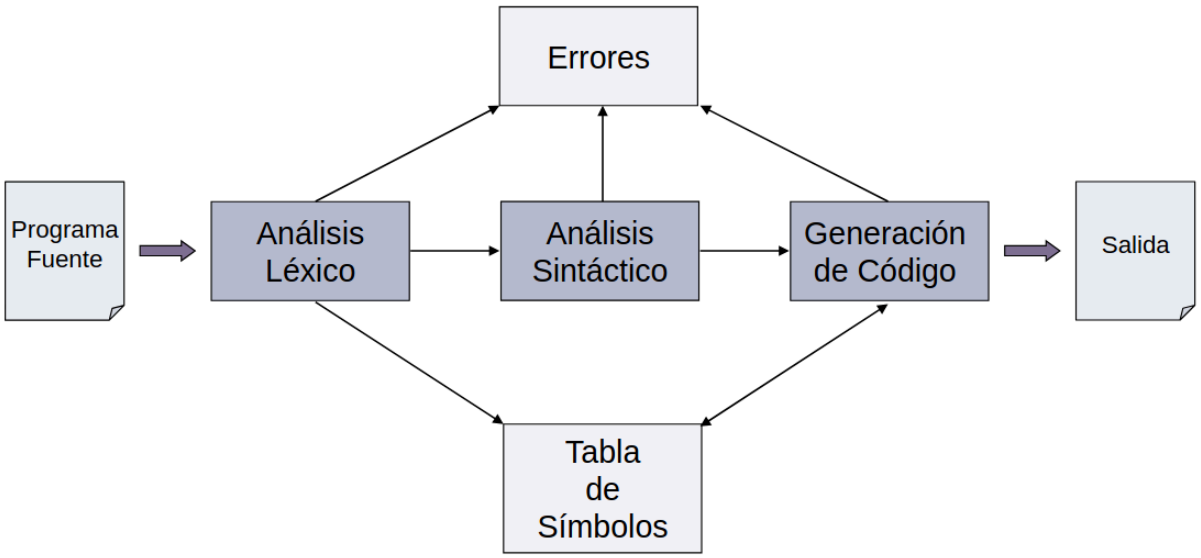

Imagen 1: etapas de un compilador.

A lo largo del informe, se detalla el desarrollo e implementación de las primeras dos etapas: el **análisis léxico** y el **análisis sintáctico**.

Consideraciones

Varias de las imágenes presentadas a lo largo del informe pueden encontrar su calidad degradada, puesto a que son grandes, contienen muchos detalles y el editor de documentos empleado no cuenta con soporte para SVG. En el código fuente, específicamente en la carpeta resources/images/, se adjuntan las imágenes en resolución completa, tanto *png* como *svg*, para poder visualizarlas de forma detallada.

Temas Particulares

A continuación, se describen brevemente los temas asignados al grupo.

Tema 2

Enteros sin signo de 16 bits. Se deben incorporar al lenguaje constantes con valores entre 0 y $2^{16} - 1$. Estas se escriben como una secuencia de dígitos seguidos del sufijo `UI`. Adicionalmente, se debe incorporar a la lista de palabras reservadas la palabra `uint`.

Tema 5

Punto flotante de 32 bits. Se deben incorporar al lenguaje números reales con signo y parte exponencial. Estos únicamente podrán ser utilizados como constantes, es decir, no podrán declararse variables de este tipo. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra `F`, seguido obligatoriamente del signo del exponente y su valor. El '.' es obligatorio. Solo una de las partes **puede** estar ausente en una misma constante. Es decir, si falta la parte entera, debe estar presente la decimal, y viceversa.

El rango admitido para los números de punto flotante es el intervalo (1.17549435F-38, 3.40282347F+38) para los positivos y (-3.40282347F+38, -1.17549435F-38) para los negativos, además del valor 0.0.

Tema 7

Cadenas de una línea. Se debe incorporar al lenguaje cadenas que no puedan ocupar más de una línea. Estas deben estar delimitadas por comillas dobles. Ejemplo: "Hola Mundo".

Tema 14

do while. Se debe incorporar al lenguaje la siguiente estructura: `do <bloque_de_sentencias_ejecutables> while ( <condicion> );`, donde: `<condicion>` posee la misma definición que la condición de las sentencias IF; y `<bloque_de_sentencias_ejecutables>` podrá contener una sentencia, o un grupo de sentencias ejecutables delimitado por llaves.

Como consecuencia de esta estructura, se debe incorporar a la lista de palabras reservadas las palabras `do` y `while`.

Tema 17

Asignaciones que pueden tener menor número de elementos del lado izquierdo. Se deben reconocer asignaciones múltiples, que permitan una lista de variables, separadas por coma

(",") del lado izquierdo de la asignación, y una lista de constantes separadas por coma (",") del lado derecho. El operador utilizado para la asignación será "=" y el número de constantes del lado izquierdo puede ser menor al de variables en el lado derecho.

Tema 22

Prefijado obligatorio. Se debe incorporar la posibilidad de utilizar, en cualquier lugar donde pueda participar una variable, un prefijado que indicará la unidad a la que pertenece. Este debe escribirse como un identificador seguido de un "." precediendo al nombre de la variable: `ID.ID`.

Este prefijado se utilizará para indicar la unidad a la que pertenece la variable, en caso de que no esté declarada localmente. Su obligatoriedad y uso correcto se chequeará en la etapa 3 del TP.

Tema 26

Copia-Valor-Resultado. Se debe agregar a la lista de palabras reservadas la palabra `cvr`. En la declaración de la función, antes de cada parámetro formal, se **podrá** indicar, mediante una palabra reservada, la semántica del pasaje de ese parámetro. Los casos posibles son:

- `<parametro_formal> <tipo> ID`, donde se usa la semántica por defecto.
- `<parametro_formal> cvr <tipo> ID`, donde se usa la semántica de copia-valor-resultado.

Tema 27

Expresiones lambda en línea. Se debe incorporar al lenguaje la definición y uso de expresiones lambda que pueden ser usadas en línea. La sintaxis de esta es: `<parámetro><cuerpo><argumento>`.

Estas expresiones permiten un solo parámetro, que se escribirá entre paréntesis con la estructura `<tipo> ID`. El cuerpo será un bloque de sentencias ejecutables delimitado por llaves. El argumento podrá ser un identificador o una constante. Este debe ir entre paréntesis. Ejemplo: `(uint A) { if (A > 1UI) print ("¡Hola!); } (3I)`.

Tema 30

Conversiones implícitas.Se explicará y resolverá en trabajos prácticos 3 y 4.

Tema 33

Comentarios multilínea. Se debe incorporar al lenguaje la posibilidad de escribir comentarios que puedan ocupar más de una lista. Estos deben comenzar con `##` y terminar con `##`.

Decisiones de Diseño e Implementación

El compilador fue desarrollado en Java. El foco durante todo su desarrollo se halló ubicado en un balance entre la *performance* y, por sobre todo, la legibilidad, dado que es un proyecto universitario que es evaluado por docentes.

El código se reparte principalmente en tres directorios: `Common`, `Lexer` y `Parser`. A continuación, se explica brevemente el contenido de cada uno de ellos.

`Common`

Contiene las clases comunes a varias etapas del compilador, tales como `Symbol`, `SymbolTable`, `Token` y `TokenType`.

La tabla de símbolos es implementada mediante un `HashMap<String, Symbol>`. La clave del mapa es el lexema, mientras que el valor es una instancia de la clase `Symbol`, la cual puede almacenar información relevante acerca del símbolo, tales como: el tipo de dato, el anidamiento, entre otros contenidos que se expandirán y utilizarán mejor en futuras etapas.

`Lexer`

Contiene las clases correspondientes al análisis léxico.

- `DataManager`: Es el encargado de realizar la carga de la matriz de transición de estados y de la matriz de acciones semánticas. Ambas son implementadas como matrices. En el caso de la matriz de transición de estados, esta es una matriz de enteros. Por otro lado, la matriz de acciones semánticas es una matriz de arreglos de acciones semánticas. El orden de las acciones semánticas en estos arreglos se corresponde con su orden de ejecución.
- `Lexer`: Es el analizador léxico en sí. Se encarga de ir leyendo el archivo, cargado a memoria, y transitar por los estados correspondientes a la vez que se invocan las respectivas acciones semánticas de la transición. En caso de transitar a un estado de error, es el encargado de dar aviso a la implementación del error léxico correspondiente para que ejecute su acción particular. Esta clase utiliza las matrices cargadas en `DataManager`.

Adicionalmente, hay dos subdirectorios: `errors` y `actions`. En el primero, se especifica una interfaz que define los métodos que debe tener toda clase capaz de manejar errores léxicos: `LexicalError`. En el segundo, se define una interfaz en la cual se especifica el método que debe tener toda implementación de una acción semántica: `SemanticAction`. A su vez, dentro de cada uno de estos, hay un subdirectorio `implementations` con las implementaciones correspondientes, las cuales serán descritas durante el desarrollo del análisis léxico.

Parser

Contiene las clases y archivos correspondientes al análisis sintáctico.

- `gramatica.y`: Archivo en el que se define la gramática del lenguaje, a la vez que código adicional.
- `Parser.java`: Es la clase que implementa el *parser*. Es creada automáticamente mediante la herramienta *yacc*. No obstante, también contiene código adicional que se definió en el archivo anterior.
- `ParserVal.java`: Esta clase también es creada automáticamente por la herramienta *yacc* y sirve como medio mediante el cual el `Lexer` es capaz de pasarle la entrada a la tabla de símbolos correspondiente al *token* al `Parser`.
- `y.output`: Generado automáticamente por la herramienta *yacc*. Contiene el autómata definido a partir de la gramática.

Utilities

Hay un directorio adicional, el cual es `utilities`. Dentro suyo, se halla una clase de utilidad: `MatrixExporter`. Esta, si bien está en el código del proyecto, no forma parte de manera directa de él. Su ubicación se justifica en la necesidad de importaciones de clases del proyecto. Se encarga de convertir las matrices ubicadas en `DataManager` a archivos `csv`, para que, utilizando librerías de Python, luego se puedan renderizar a las imágenes que se presentan en este informe. El objetivo de esta clase es mantener la consistencia entre la matriz en código y la matriz en forma de visualización.

Adicionalmente, dentro del directorio, puede hallarse una clase `Printer`, implementada con el objetivo de mejorar la estética de la salida.

Análisis Léxico

Diagrama de Transición de Estados

A continuación, se presenta el diagrama de transición de estados:

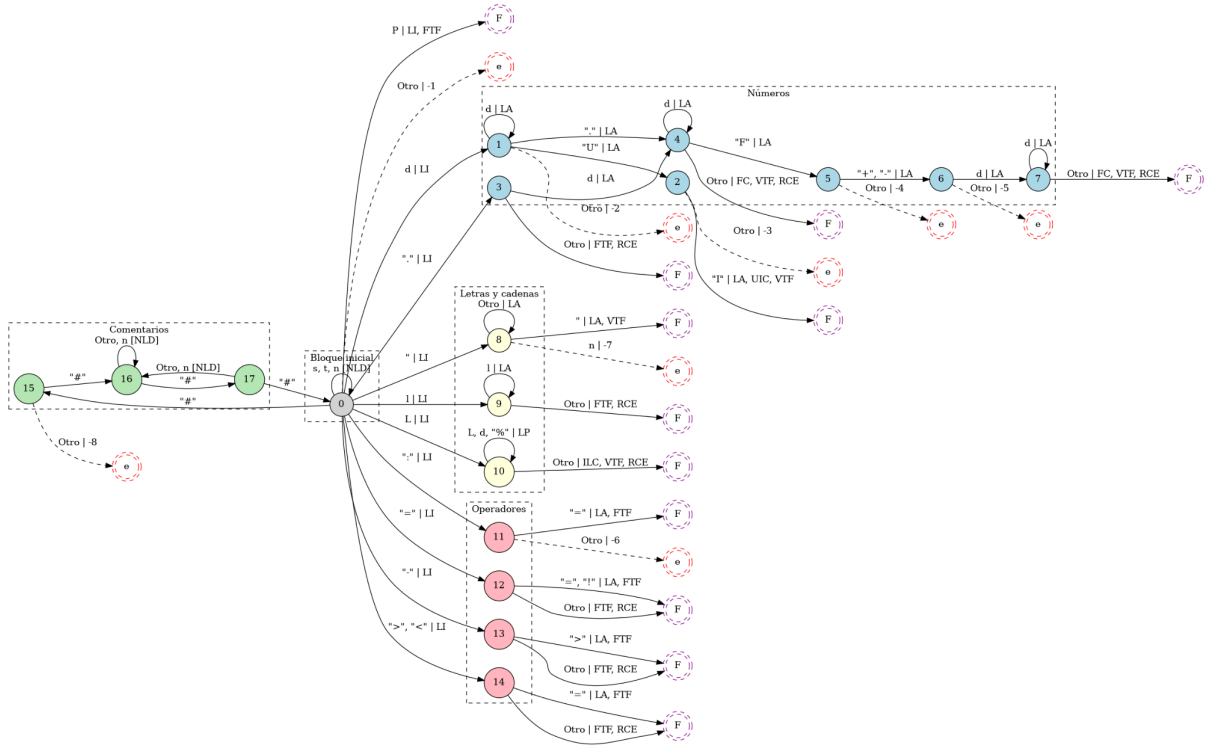

Imagen 2: diagrama de transición de estados. A considerar, el estado de error (sumidero) es uno solo, al igual que también lo es el estado de aceptación. Se duplicaron para poder presentar de forma más ordenada el autómata.

Donde se define:

- `L`: conjunto de todos los carácteres correspondientes a letras mayúsculas.
- `l`: conjunto de todos los carácteres correspondientes a letras minúsculas.
- `d`: conjunto de carácteres correspondientes a dígitos (0 a 9).
- Los carácteres específicos serán indicados entre comillas simples (" y "), exceptuando por el carácter correspondiente a estas (`'`), que será indicado en solitario, ya que indicarlo entre comillas afectaría a la claridad del autómata (`"'"`).
- `s`: indica un espacio en blanco.
- `t`: indica una tabulación.
- `n`: indica un salto de línea.
- `P`: conjunto de símbolos monocarácter. `P = {'+', '*', '/', '(', ')', '{', '}', ';', ','}`.
- `Otro`: representa a un carácter que no está comprendido en los demás arcos.

Se intentó atenerse a la convención usada por la cátedra. No obstante, si bien esta se halló de utilidad para ejemplos concretos y pequeños, ante autómatas de mayor tamaño, se consideró que generaba demasiado desorden. Por consiguiente, se propone la convención mencionada con anterioridad, junto a las particularidades relacionadas a las acciones semánticas que se mencionan en breve.

A continuación, se presenta la matriz de transición de estados resultante del diagrama de transición de estados anteriormente presentado:

	L	l	d	U	I	.	F	-	P	"	:	=	!	>	<	%	#	n	s	t	Otro
0	10	9	1	10	10	3	10	13	18	8	11	12	-1	14	14	-1	15	0	0	0	-1
1	-2	-2	1	2	-2	4	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
2	-3	-3	-3	-3	18	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3	-3
3	18	18	4	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
4	18	18	4	18	18	18	5	18	18	18	18	18	18	18	18	18	18	18	18	18	18
5	-4	-4	-4	-4	-4	-4	-4	6	6	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4
6	-5	-5	7	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5
7	18	18	7	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
8	8	8	8	8	8	8	8	8	8	18	8	8	8	8	8	8	8	-7	8	8	8
9	18	9	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
10	10	18	10	10	10	18	10	18	18	18	18	18	18	18	18	10	18	18	18	18	18
11	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	-6	18	-6	-6	-6	-6	-6	-6	-6	-6	-6
12	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
13	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
14	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
15	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	-8	16	-8	-8	-8	-8
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	17	16	16	16	16
17	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	0	16	16	16	16

Imagen 3: matriz de transición de estados.

Acciones Semánticas

Junto a algunos arcos, se indica, luego del símbolo o conjunto de símbolos y separado por una `|`, el conjunto de acciones semánticas que se aplican. En caso de aplicar la acción semántica a solo uno de los símbolos en el listado, la acción semántica es especificada entre corchetes al lado del símbolo. Por ejemplo, `n [NLD]` en el arco del estado 17 al estado 16. El orden de las acciones semánticas no es arbitrario, sino que corresponde al orden en el que estas serán ejecutadas.

Si bien la presencia de múltiples acciones semánticas para un mismo arco puede parecer engorrosa, se apuntó durante todo el proceso a un **autómata autoexplicativo**. Para explicar este concepto, se presenta la alternativa: una única acción semántica por arco. Esta acción semántica, dentro suyo, realizaría varias operaciones. Posiblemente, varias de estas sean comunes a varias acciones semánticas. Como consecuencia, estas se modularizarían y pasarían a ser métodos. El resultado es que una acción semántica no realizaría otra cosa más que invocar a determinadas operaciones. Para conocer a qué operaciones invoca una determinada acción semántica, sería necesario observar el código. En contraste, se consideró que atomizando las operaciones en acciones semánticas en sí mismas y presentándose en forma de lista, existe una mayor claridad y se facilita, observando el autómata y conociendo el propósito de cada acción, la comprensión acerca de lo que se realiza en cada transición sin la necesidad de recurrir al código. Adicionalmente, durante el desarrollo, esto facilita el *debugging*.

Las acciones semánticas definidas, en orden alfabético, son:

- `FixedTokenFinalizer` (FTF): Detecta el tipo de token de palabras reservadas y operadores (*tokens* a los que les corresponde un único lexema, los cuales denominamos ***tokens fijos***) y setea el tipo de *token* correspondiente de acuerdo a la detección. En caso de no ser válida la palabra reservada, levantará un error léxico: `UnknownToken`.
- `FloatChecker` (FC): Valida que el flotante esté dentro del rango de representación válido y mejora su legibilidad (esto se explica con mayor detalle en la sección de decisiones). En caso de no estar en el rango de representación válido, lo reemplaza por `0.0`, presentando una *warning* al usuario.
- `IdentifierLengthChecker` (ILC): Valida que el identificador se halle en el rango de los 20 caracteres. En otro caso, lo trunca, presentando una *warning* en la salida.
- `LexemaAppender` (LA): Agrega el último carácter leído al string del lexema.
- `LexemaInitializer` (LI): Inicializa el string del lexema, agregando el carácter leído.
- `NewLineDetected` (NLD): Incrementa el contador de números de línea del programa.
- `ReturnCharacterToEntry` (RCE): Decrementa la referencia al siguiente carácter a leer (devolviendo así el último carácter leído a la entrada). Especialmente útil cuando se debe volver a leer un carácter.
- `UintChecker` (UIC): Se eliminan los *leading zeros* (ceros a la izquierda) y se verifica que el número está dentro del rango de `uint`. Si no lo está, se le asigna el máximo valor en el rango.
- `VariableTokenFinalizer` (VTF): Detecta el tipo de token de *tokens* a los que les corresponde más de un lexema (identificadores, constantes y cadenas, a los que denominamos ***tokens variables***) y setea el tipo de *token* correspondiente de acuerdo a la detección. Adicionalmente, da de alta la entrada para dicho *token* en la tabla de símbolos.

En la siguiente imagen, es posible observar la matriz de acciones semánticas resultante:

	L	l	d	U	I	.	F	-	P	"	:	=	!	>	<	%	#	n	s	t	Otro
0	LI	LI	LI	LI	LI	LI	LI	LI	LI, FTF	LI	LI	LI		LI	LI			NLD			
1	LA	LA	LA	LA		LA															
2					LA, UIC, VTF																
3	FTF, RCE	FTF, RCE	LA	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE
4	FC, VTF, RCE	FC, VTF, RCE	LA	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	LA	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE
5								LA	LA												
6			LA																		
7	FC, VTF, RCE	FC, VTF, RCE	LA	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE	FC, VTF, RCE
8	LA	LA	LA	LA	LA	LA	LA	LA	LA	LA, VTF	LA	LA	LA	LA	LA	LA	LA	NLD	LA	LA	LA
9	FTF, RCE	LA	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE
10	LA	ILC, VTF, RCE	LA	LA	LA	ILC, VTF, RCE	LA	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	LA	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE	ILC, VTF, RCE
11												LA, FTF									
12	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	LA, FTF	LA, FTF	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE
13	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	LA, FTF	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE
14	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	LA, FTF	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE	FTF, RCE
15																					
16																		NLD			
17																		NLD			

Imagen 4: matriz de acciones semánticas.

Errores Léxicos

- `InvalidSymbol` (-1). Ocurre cuando se está en el estado 0 y el símbolo no corresponde a un arco válido.
- `UndeterminedNumber` (-2). Ocurre cuando se está en el estado 1 y se detecta un carácter diferente a un dígito, un punto (que indicaría que se está frente a un flotante) o el inicio del sufijo de un entero. Es decir, este error ocurre cuando se detecta un número que no corresponde con el formato válido de un entero sin signo ni de un flotante. No es posible determinar, a partir de un número, como "11", si la intención del usuario es la de escribir un entero (y se olvidó de escribir el sufijo) o si su intención es la de escribir un flotante (y se olvidó, por lo tanto, del punto). Como corrección a este error, se asume que la intención del usuario fue escribir un entero y se completa el lexema con el sufijo `UI`.
- `BadUISuffix` (-3). El error ocurre cuando se escribe un número entero, junto al carácter `U`. Sin embargo, el último carácter correspondiente al sufijo de los enteros sin signo está faltante (`I`). Se asume que el usuario olvidó este carácter, por lo que se agrega, mostrando una *warning*, y se retorna el *token* identificado como constante entera.
- `NoExponentSign` (-4). Este error ocurre cuando no se especifica el signo del exponente. Se completa el lexema, asumiendo que el exponente es positivo y presentando una *warning*.
- `NoExponent` (-5). Números decimales en los que se especifica el símbolo "F" y un operador de suma ("+", "-"), pero no se especifica el valor del exponente. Por ejemplo, "1.2F+". Se completa el valor del exponente con cero y se informa esta asunción mediante una *warning*.
- `InvalidAssignmentOperator` (-6). Se escribe ":" dos puntos solo. El único símbolo del lenguaje que contiene ":" es el operador de asignación ":=". Por consiguiente, se asume que el usuario tenía la intención de escribir dicho operador y se autocompleta, informando esta asunción mediante una *warning*.
- `NewLineInString` (-7). Como se aclaró en la mención de los temas, en particular, en el tema 7, las cadenas de nuestro lenguaje no pueden tener saltos de línea. Este error ocurre en caso de que eso no se cumpla, es decir que, durante la escritura de una cadena, se utilice un salto de línea. Lo que se realiza en tal caso es descartar el salto de línea, a la vez que se muestra un mensaje de error.
- `BadCommentInitialization` (-8). Como se hizo mención en los temas, específicamente, en el tema 33, los comentarios de nuestro lenguaje son multilínea y se hallan delimitados por la secuencia de caracteres `##`. De venir un único `#`, este se descarta.

  En el peor caso, si fue un intento de comentario, se considerará un comentario desde el fin de este hasta el inicio válido de otro comentario, descartando así sentencias

válidas del programa. En el mejor de los casos (el carácter se escribió como un error), al ser este descartado, se analizará el código como si nunca hubiese sido escrito, correspondiendo con la intención del usuario.

Esto se evidencia mejor en el siguiente ejemplo:

```
1. # Inicio de comentario
2. Texto en el comentario
3. Fin de comentario ##
4.
5. uint X;
```

Si se descarta el primer `#`, los *tokens* de la sentencia `uint X;` serían descartados, ya que se consideraría parte del comentario iniciado al final de la línea 3. Esto podría continuar, en el peor caso, hasta el final del archivo.

Fueron consideradas otras alternativas, como: contar los `#` que siguen y determinar si fue un intento de comentario o un error. Sin embargo, el analizador léxico tendría que leer todo el archivo por adelantado y, además, podría haber `#` dentro de los propios comentarios o haber varios comentarios mal escritos. Por eso, ante la dificultad de poder determinar la intención del usuario, se decidió simplemente descartar el carácter, asumiendo la consecuencia de esta acción: partes del código cuya intención no eran las de funcionar como un comentario ahora serían consideradas como tal, y viceversa.

- `UnknownToken`. No posee presencia en el grafo, ya que solo se accede a él durante la acción semántica `FixedTokenFinalizer`. La intención de esta clase, si bien podría parecer innecesaria, es concentrar los mensajes de error léxicos únicamente en implementaciones de la interfaz `LexicalError`. De esta forma, se tienen por un lado, las acciones semánticas, capaces de mostrar advertencias y ejecutar acciones reparadoras, y los errores léxicos, capaces de detectar errores y mostrar un mensaje personalizado al respecto.

Decisiones

Se consideró que agrupar todas las decisiones en una única sección no era conveniente, ya que muchas de ellas requieren de explicaciones previas o son mejor explicadas en su contexto (por ejemplo, las decisiones tomadas en los errores léxicos). Por consiguiente, como se habrá notado, varias ya fueron explicadas durante el desarrollo de este. Sin embargo, existen decisiones más generales a las que aún no se les dio un lugar. Estas se detallan a continuación:

Desambiguación

En el reconocimiento de varios operadores juntos, se decidió optar por una regla de desambiguación de lectura a partir de la izquierda. Por ejemplo, de detectarse en el código `<==`, los *tokens* detectados podrían ser `<=` y `=` o, `<` y `==`. De acuerdo a la regla de desambiguación propuesta, la primera opción es la considerada por el analizador léxico.

Esta regla no solo aplica a operadores mezclados, sino también a posibles confusiones entre constantes. Por ejemplo, de recibir algo como `31.2UI`, no es del todo claro si el usuario buscaba escribir un entero o un flotante. Sin embargo, aplicando la desambiguación, este sería interpretado como una constante flotante `31.2` y un identificador `UI`.

Los casos en los que esta regla podría participar son varios y se consideraron otras alternativas para casos particulares. Por ejemplo, para el caso anterior de `31.2UI`, se consideró la posibilidad de realizar la asunción de que se trataba de un entero y descartar la parte decimal. Sin embargo, los casos son variados y adivinar la verdadera intención del usuario, además de ser una tarea ardua, implicaría un aumento significativo en la cantidad de código y en su complejidad. Por esta misma razón, se realizaron asunciones en los segmentos de código en los que se consideró que era fácil o conveniente hacerlas (por ejemplo, en `InvalidAssignmentOperator`, asumir, al encontrar ":", que se quiso escribir el operador de asignación y completar con "="), y se optó por no hacerlas en aquellos casos en los que la decisión no era tan clara o no era tan fácil inclinarse por un lado de la balanza.

Flotantes

Con motivos de legibilidad, las constantes flotantes cuya parte entera o flotante se encuentre faltante son completadas. Es decir, en caso de identificar una constante flotante escrita por el usuario como `0.`, esta se almacenará en la tabla de símbolos como `0.0`. Adicionalmente, de detectar un flotante que puede ser escrito de forma más compacta en notación científica, por ejemplo, `0.000001`, se hará la transformación a dicha notación.

Guardado en la Tabla de Símbolos

Considerando las recomendaciones de la cátedra, se tomó la decisión de guardar las constantes en la tabla de símbolos de forma diferente a cómo son escritas en el lenguaje. Esto se realizó con el objetivo de facilitar el futuro proceso de generación de código. Con esto en cuenta, se estableció que:

- las constantes enteras serían guardadas sin prefijo, por ejemplo, en lugar de guardarse `4UI`, almacenarse `4`; y
- las constantes flotantes, con la letra `e` en lugar de `F`, por ejemplo, `12e-3` en lugar de `12F-3`.

Análisis Sintáctico

El análisis sintáctico fue implementado usando la herramienta Byacc/J. A continuación, se describirán la lista de no-terminales, los errores sintácticos reconocidos por el compilador y las decisiones más importantes tomadas durante su desarrollo.

No Terminales

Los no-terminales que se mencionan a continuación conforman la gramática del lenguaje. Muchos de ellos representan las estructuras posibles y admisibles del lenguaje. Adicionalmente, estos pueden contener reglas de error, producto de la extensión de la gramática para abarcar la posibilidad de errores semánticos. Sin embargo, otros no-terminales cumplen el único propósito de abarcar casos de error. Estos últimos se hallan subrayados para indicar que no forman parte de la gramática admisible.

La lista de no-terminales, junto a una breve descripción acerca de ellos, es la siguiente:

- `programa`. Regla de inicio. Marca el inicio del programa y su estructuración completa. Además, abarca errores respectivos a este.
- `programa_sin_nombre`. Establece la posibilidad de que un programa pueda no tener un nombre. Útil para mostrar un mensaje personalizado acerca de este error..
- `cuerpo_programa`. Describe la estructura válida para el cuerpo de un programa.
- `cuerpo_programa_recuperación`. Abarca estructuras alternativas al cuerpo del programa. Por ejemplo, la falta de las llaves delimitadoras.
- `lista_llaves_apertura`. Maneja el error de que un programa cuente con múltiples llaves de apertura.
- `lista_llaves_cierre`. Maneja el error de que un programa cuente con múltiples llaves de cierre.
- `conjunto_sentencias`. Utilizado para representar una secuencia de sentencias, tanto ejecutables como declarativas.
- `sentencia`. Creado con motivos de legibilidad y simplificación de `conjunto_sentencias`. Diferencia dos tipos de sentencias: `sentencia_declarativa` y `sentencia_ejecutable`.
- `sentencia_declarativa`. Abarca los distintos no terminales asociados a sentencias declarativas en el programa (declaración de variables y declaración de función).
- `punto_y_coma_opcional`. Como se mencionará más adelante en las decisiones, se decidió permitir la opcionalidad del ‘`;`‘ al final del cuerpo de las funciones. Este no-terminal abarca la posibilidad de que haya, o no, un punto y coma.
- `cuerpo_ejecutable`. Permite la posibilidad de colocar una única sentencia ejecutable o un bloque de estas. Especialmente útil en sentencias de control, como el `if`, las cuales permiten en su cuerpo una única sentencia o un bloque de estas.

- `bloque_ejecutable`. Conjunto de sentencias ejecutables delimitadas por llaves.
- `conjunto_sentencias_ejecutables`. Secuencia recursiva a izquierda de sentencias ejecutables.
- `sentencia_ejecutable`. Abarca los distintos no terminales asociados a sentencias ejecutables en el programa (por ejemplo, la invocación a funciones, sentencias de control, la sentencia `print`, etc.).
- `sentencia_control`. Separado por motivos de legibilidad. Abarca los distintos no-terminales asociados a sentencias de control (`if` y `while`).
- `declaracion_variables`. Representa la estructura válida de una declaración de variables, al igual que errores asociados a ellas.
- `lista_variables`. Utilizado en `declaracion_variables`. Permite el listado recursivo a izquierda de variables en la declaración.
- `asignacion_simple`. Establece la estructura válida de una asignación simple, al igual que errores asociados a ella.
- `asignacion_multiple`. Establece la estructura válida de una asignación múltiple, al igual que sus errores asociados.
- `asignacion_par`. Regla recursiva. Agrega variables y constantes desde el centro hacia afuera.
- `variable_con_coma`. Establece la posibilidad de una variable precedida por una coma o una variable sin coma. En este último caso, se presenta un error.
- `constante_con_coma`. Establece la posibilidad de colocar una constante seguida de una coma o una constante sin esta. En este último caso, se presenta un error.
- `lista_constantes`. Lista de constantes que permite, en una asignación múltiple. que haya mayor cantidad de constantes (elementos del lado derecho) que variables (elementos del lado izquierdo).
- `expresion`, `termino` y `factor`. Tomados de la teoría. Permiten la construcción de expresiones aritméticas respetando la precedencia de operadores.
- `termino_simple` y `factor_simple`. Su justificación y explicación se detalla en la sección de decisiones.
- `operador_suma`. Representa a los operadores de suma: ‘+’ y ‘-’.
- `operador_multiplicacion`. Representa a los operadores de multiplicación: ‘*’ y ‘/’.
- `constante`. Representa una constante, positiva o negativa.
- `variable`. Representa una variable, con o sin prefijado.
- `condicion`. Establece la estructura válida de una condición, al igual que sus reglas de error.
- `cuerpo_condicion`. Representa el cuerpo de una condición, constituido por dos expresiones y un operador de comparación, y sus reglas de reglas asociadas.

- `comparador`. Representa todos los operadores de comparación válidos en el lenguaje, al igual que reglas de error correspondientes a casos particulares (por ejemplo, colocar '`=`' en lugar de '`==`' es un error específico y común).
- `if`. Establece la estructura válida de construcción de la sentencia `if` en el lenguaje, al igual que sus reglas de error.
- `rama_else`. Creada con motivos de legibilidad. Establece la estructura de una posible rama `else` en una sentencia `if`.
- `do_while`. Establece la estructura válida de una sentencia `do-while`, al igual que sus casos de error asociados.
- `cuerpo_do`. Representa la estructura admisible del cuerpo de una sentencia do-while.
- `cuerpo_do_recuperacion`. Contempla las reglas de error del cuerpo de la sentencia do-while.
- `fin_cuerpo_do`. Creada con motivos de legibilidad. Representa el fin admisible de una sentencia do-while.
- `declaracion_funcion`. Establece la estructura válida de una declaración de función, al igual que su caso de error asociado.
- `cuerpo_funcion`. Representa el cuerpo de una función, incluidos sus errores.
- `cuerpo_funcion_admisible`. Representa únicamente el cuerpo admisible de una función. Se propuso con el objetivo de únicamente notificar la detección semántica de una declaración de función en caso de que el cuerpo sea correcto.
- `conjunto_parametros`. Representa a una lista de parámetros, delimitados por paréntesis.
- `lista_parametros`. Representa a una lista, recursiva a izquierda, de parámetros.
- `parametro_vacio`. Contempla la posibilidad de que exista un parámetro faltante en una declaración de función.
- `parametro_formal`. Establece la estructura válida de un parámetro formal en la declaración de una función, al igual que sus errores asociados.
- `semantica_pasaje`. Contempla las posibles semánticas de pasaje de parámetros.
- `sentencia_retorno`. Describe la estructura válida de la sentencia de retorno, a la vez que sus posibles errores asociados.
- `invocacion_funcion`. Establece la estructura válida de una invocación de función.
- `lista_argumentos`. Representa a un argumento a una lista de estos.
- `argumento`. Establece la estructura admisible de un argumento, al igual que su caso de error asociado.
- `impresion`. Describe la estructura válida de una impresión o sentencia 'print', al igual que sus casos de error asociados.

- `imprimible`. Representa la estructura válida de un imprimible (`elemento_imprimible` entre paréntesis). Solamente se muestra la detección de una impresión si se reduce por este no-terminal y no, por el anterior.
- `imprimible_recuperacion`. Contempla la posibilidad de un imprimible no admisible y sus mensajes de error correspondientes.
- `elemento_imprimible`. Elemento capaz de ser imprimido (cadena o expresión).
- `lambda`. Representa la estructura válida de una sentencia 'lambda', al igual que sus errores asociados.
- `argumento_lambda`. Abarca la construcción correcta de un argumento lambda. Únicamente se presenta la detección de expresión 'lambda' ante este no-terminal.
- `argumento_lambda_recuperacion`. En caso de entrar por esta regla, la expresión lambda contiene un argumento no válido y, por lo tanto, no se presenta su detección semántica.
- `parametro_lambda`. Creado con motivos de legibilidad. Representa la estructura admisible de un parámetro *lambda*.

Errores Sintácticos

Los errores sintácticos que el compilador reconoce se describen a continuación, categorizados según las sentencias que abarcan. Se hallan resaltados (en verde y azul) aquellos solicitados como mínimos por la cátedra. En su mayoría, únicamente se brindan los mensajes de error, pues estos son autoexplicativos acerca de lo que abarcan.

Generales

- "Las sentencias del programa deben estar delimitadas por llaves."
- "El programa requiere de un nombre."
- "Inicio de programa inválido. Se encontraron, previo al nombre del programa, sentencias."
- "Se llegó al fin del programa sin encontrar un programa válido."
- "Se encontraron múltiples llaves al comienzo del programa". En caso de que el programa comience con múltiples llaves de apertura.
- "Se encontraron múltiples llaves al final del programa". En caso de que el programa termine con múltiples llaves de apertura.
- "El programa no posee ninguna sentencia".
- "El programa no posee ningún cuerpo".
- "Cierre inesperado del programa. Verifique llaves '{...}' y puntos y coma ';' faltantes..
- "El cuerpo de la sentencia no puede estar vacío." Por ejemplo, en el caso de que un `if`, `do-while` o `else` tenga un cuerpo con ninguna sentencia dentro.
- "El archivo está vacío."

Declaración de Variables

- "La declaración de variable debe terminar con ';'.".

- "La declaración de variables debe terminar con ';'.". La diferencia con el anterior es que este abarca el caso en el que se especifiquen varias variables.
- "Declaración de variables inválida".
- "La declaración de variables y la asignación de un valor a estas debe realizarse en dos sentencias separadas." Esto forma parte de una decisión que se tomó: no permitir declaraciones y asignaciones en una misma sentencia.
- "Se encontraron dos variables juntas sin separación. Inserte una ',' entre a y b.", donde a y b son los lexemas asociados a las variables.

Asignación Simple

- "Las asignaciones simples deben terminar con ';'."
- "Error en asignación simple. Se esperaba un ':=' entre la variable y la expresión."
- "Asignación simple inválida."

Asignación Múltiple

- "La asignación múltiple debe terminar con ';'."
- "Se encontraron dos constantes juntas sin una coma de separación. Inserte una ',' entre a y b.", donde a y b son constantes.
- "Falta coma antes de variable 'a' en asignación múltiple.", donde a es una variable.
- "Falta coma luego de constante 'a' en asignación múltiple.", donde a es una constante.

Expresiones

- "Falta de operando en expresión luego de a b.", donde a es un operando y b es uno de los cuatro operadores válidos en el lenguaje.
- "Falta de operador entre operandos b y a.", donde a y b son operandos.

Condición

- "Falta apertura de paréntesis en condición."
- "La condición no puede estar vacía."
- "La condición debe ir entre paréntesis."
- "Falta cierre de paréntesis en condición."
- "Falta de comparador en comparación."
- "Se esperaba un comparador y se encontró el operador de asignación '='. ¿Quiso colocar '=='?".

Sentencia IF

- "La sentencia IF debe terminar con ';'."
- "La sentencia IF debe finalizar con 'endif'."
- "Sentencia IF inválida."

Sentencia DO-WHILE

- "La sentencia 'do-while' debe terminar con ';'."
- "Sentencia 'do-while' inválida."
- "Debe especificarse un cuerpo para la sentencia do-while."
- "Falta 'while'."

Declaración de Función

- "La función requiere de un nombre."
- "El cuerpo de la función no puede estar vacío."
- "Toda función debe recibir al menos un parámetro."
- "Se halló un parámetro formal vacío."
- "Falta de nombre de parámetro formal en declaración de función."
- "Falta de tipo de parámetro formal en declaración de función."
- "Semántica de pasaje de parámetro inválida."

Invocación de Función

- "La invocación a función debe terminar con ';'."
- "Falta de especificación del parámetro formal al que corresponde el parámetro real."

Sentencia RETURN

- "La sentencia 'return' debe terminar con ';'."
- "El retorno no puede estar vacío."
- "El resultado a retornar debe ir entre paréntesis."
- "Sentencia 'return' inválida."

Sentencia PRINT

- "La sentencia 'print' debe finalizar con ';'."
- "La sentencia 'print' requiere de al menos un argumento."
- "El imprimible debe encerrarse entre paréntesis."
- "La sentencia 'print' requiere de un argumento entre paréntesis."

Expresión LAMBDA

- "La expresión 'lambda' debe terminar con ';'."
- "Falta delimitador de cierre en expresión 'lambda'."
- "Faltan delimitadores en el conjunto de sentencias de la expresión 'lambda'."
- "Falta delimitador de apertura en expresión 'lambda'."
- "El argumento de la expresión 'lambda' no puede estar vacío."
- "La expresión 'lambda' requiere de un argumento entre paréntesis."

Errores No Abordados

Como se habrá notado, hay tres errores subrayados en azul. Dos de ellos son: “Las asignaciones simples deben terminar con ';'.” y "Falta cierre de paréntesis en condición.". Estos representan los errores mínimos que no fue posible abordar.

Las reglas de error de estos errores comparten algo en común: ambos, antes del *token* `error`, tienen el no-terminal `expresion`. Por ejemplo, la regla para el primer error subrayado es la siguiente:

```
variable DASIG expresion error
```

Y del segundo es la que sigue:

```
‘(‘ cuerpo_condicion error
```

donde las reglas de `cuerpo_condicion` finalizan con el no-terminal `expresion`. Es decir, la secuencia `expresion error` está implícita mediante reducciones.

El formato usado en estas dos reglas no varía en lo absoluto con el formato utilizado en las demás reglas en las que se notifica un punto y coma faltante. En todas ellas se utiliza el *token* `error` y el funcionamiento es el esperado. La única diferencia, como se mencionó, es que el *token* de error es precedido, en estos casos, por el no-terminal `expresion`.

Realizando un *debugging*, se halló que, en estas situaciones, `termino` nunca se reduce a `expresion` cuando este último no-terminal lo prosigue, en una regla, un *token* error. Como resultado, no hace *match* con las reglas definidas y no presenta el mensaje correspondiente.

Se consultó con profesores de la cátedra acerca del por qué de este error y no se halló explicación ni solución. De igual forma, se intentó refactorizar varias veces la gramática y probar distintas alternativas para solucionarlo. No obstante, ninguna de las alternativas probadas tuvo éxito.

El tercer error subrayado es: “La condición debe ir entre paréntesis.”. A pesar de múltiples intentos, no fue posible implementarlo exitosamente, debido a numerosos *shift/reduce* cuya solución requería descartar la detección de otros errores mínimos o estructuras indispensables en el programa.

Como compensación a estos errores, se decidieron abordar varios otros que no fueron solicitados por la cátedra. Estos corresponden a los listados en la sección anterior y que no se hallan subrayados.

Decisiones

A continuación, se mencionan las decisiones más importantes tomadas con respecto a la escritura de la gramática.

Punto y Coma

El punto y coma final se requiere en todas las sentencias, exceptuando la declaración de función. Podría considerarse lógico entonces trasladarlo a una regla alta, de forma que no deba especificarse en cada una de las sentencias. Sin embargo, esto trae dos problemas:

- se pierde la posibilidad de mostrar un mensaje personalizado acerca de la sentencia en la que falta el punto y coma, como se mostró en el listado de errores sintácticos anterior; y
- se pierde la posibilidad de utilizarla como un *token* de sincronización luego de un error en la sentencia.

Con esto en cuenta, con la consecuencia de complejizar la gramática, se optó por colocar el requerimiento en cada sentencia individual.

Sentencia de Retorno

Lo ideal sería que la sentencia de retorno solo sea permitida dentro del cuerpo de una función. Esto podría, en un principio, implementarse de forma sencilla en la gramática. No obstante, una función también podría tener anidadas dentro otras estructuras, como una sentencia `if`, por ejemplo. ¿Cómo se hace la distinción, en esta etapa, entre una sentencia `if` dentro del cuerpo de una función y una fuera de este?

Una solución a este problema podría ser duplicar todos los no-terminales que pueden ser utilizados dentro del cuerpo de una función y pueden tener sentencias dentro (como la sentencia `if`). De esta forma, se diferencia entre `if` e `if_funcion`, permitiendo que los retornos solo aparezcan dentro de funciones.

Sin embargo, se consideró que esta solución conlleva ensuciar innecesariamente la gramática, pues se cree que, en el análisis semántico, esto podría resolverse de forma más sencilla. Por consiguiente, se decidió permitir, sintácticamente, la sentencia de retorno en cualquier lugar del programa. No obstante, podría reevaluarse esta decisión a futuro, una vez que se incorporen más conocimientos acerca de las etapas posteriores del compilador.

Falta de Operador y Constante Negativa

Omitiendo el error mencionado en la sección de “Errores No Abordados”, el mayor error al que se enfrentó fue un *shift/reduce*, producido por la posibilidad de un operando faltante y, al mismo, una constante negativa.

En una sentencia de la forma: `a - b`, el compilador no era capaz de decidir si la expresión era `(a) - (b)`, abarcado por la regla `expresion operador_suma termino`, o `(a) (-b)`, abarcada por la regla de error expresion termino, la cual encierra la posibilidad de falta de operador.

La solución a esto fue añadir un nuevo no-terminal, `termino_simple`, y modificar la regla de error `expresion termino` por expresion `termino_simple`. ¿Qué diferencia a `termino_simple` de `termino`? Que `termino_simple` no permite la posibilidad de constantes negativas, eliminando así la ambigüedad originada por la regla.

Punto y Coma al Final de Función

No se creyó intuitivo requerir de un punto y coma al final de la función, pues la mayoría de los lenguajes no lo requieren. Sin embargo, tampoco se consideró acertado obligar al usuario a no colocarlo, dado que todas las demás sentencias lo requieren. Por consiguiente, se permitió su opcionalidad, dejando la decisión completamente en el programador.

Salida del Programa

La salida del programa se conforma de la siguiente manera:

```
|=============================================================================================|
|                                  Resultados de la Compilación                               |
|                                      Archivo: ejemplo.uki                                   |
|=============================================================================================|


|=============================================================================================|
| 257  ID     PROGRAMA                                                                        |
| 123  {                                                                                      |
| 270  uint                                                                                   |
|=============================================================================================|
| WARNING LÉXICO: Línea 3: El identificador PALABRAHIPERLARGUISIMA excede la longitud máxima de 20 |
| caracteres. Se truncará a PALABRAHIPERLARGUISI.                                             |
|=============================================================================================|
| 257  ID     PALABRAHIPERLARGUISI                                                            |
| 59   ;                                                                                      |
|=============================================================================================|
| DETECCIÓN SEMÁNTICA: Declaración de variable.                                               |
|=============================================================================================|
| 266  print                                                                                  |
| 40   (                                                                                      |
| 259  STR    "Hola"                                                                          |
| 41   )                                                                                      |
| 59   ;                                                                                      |
|=============================================================================================|
| DETECCIÓN SEMÁNTICA: Sentencia 'print'.                                                     |
|=============================================================================================|
| 267  if                                                                                     |
| 257  ID     X                                                                               |
|=============================================================================================|
| ERROR SINTÁCTICO: Línea 7: Falta apertura de paréntesis en condición.                       |
|=============================================================================================|
| 62   >                                                                                      |
| 257  ID     B                                                                               |
| 41   )                                                                                      |
|=============================================================================================|
| DETECCIÓN SEMÁNTICA: Condición.                                                             |
|=============================================================================================|
| 257  ID     B                                                                               |
| 264  :=                                                                                     |
| 257  ID     C                                                                               |
.                                                                                             .
| 59   ;                                                                                      |
|=============================================================================================|
| DETECCIÓN SEMÁNTICA: Asignación simple.                                                     |
|=============================================================================================|
| 269  endif                                                                                  |
| 59   ;                                                                                      |
|=============================================================================================|
| DETECCIÓN SEMÁNTICA: Sentencia IF.                                                          |
|=============================================================================================|
| 125  }                                                                                      |
| 0    EOF                                                                                    |
|=============================================================================================|
| DETECCIÓN SEMÁNTICA: Programa.                                                              |
|=============================================================================================|


|=============================================================================================|
|                                    > Compilación Finalizada <                               |
|                 El programa tiene 8 líneas. Se detectaron 1 warnings y 1 errores.           |
|=============================================================================================|
```

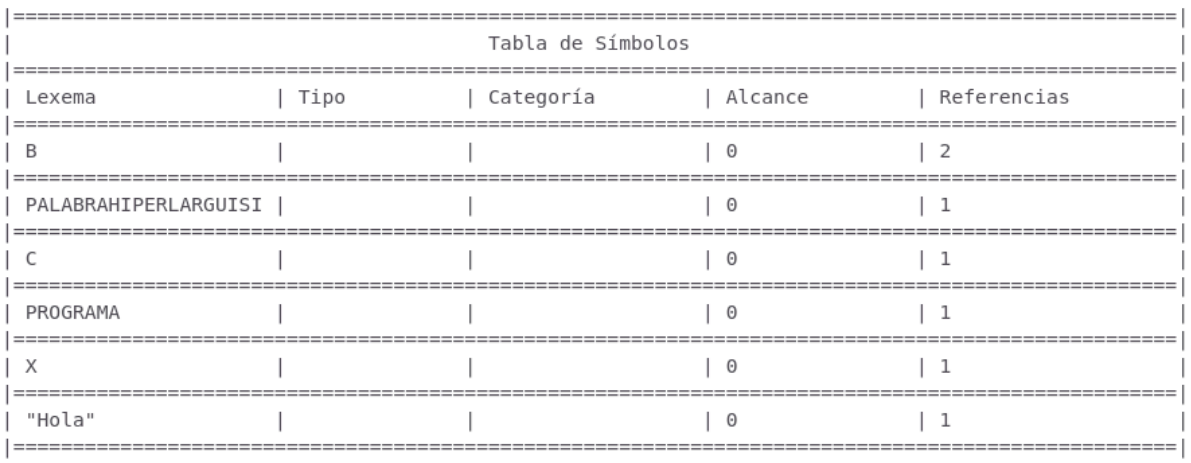

Tabla de Símbolos				
Lexema	Tipo	Categoría	Alcance	Referencias
B			0	2
PALABRAHIPERLARGUISI			0	1
C			0	1
PROGRAMA			0	1
X			0	1
"Hola"			0	1

Imagen 5: ejemplo de salida del programa.

En ella es posible observar, principalmente, un listado de *tokens*. Estos comprenden tres columnas:

- Primera columna: Código de identificación del *token*. En caso de ser un símbolo monocarácter, se corresponde con el código ASCII del carácter. En otro caso, el código es determinado por *yacc*.
- Segunda columna: Tipo de *token*. En caso de ser un símbolo monocarácter, se muestra el carácter correspondiente. En caso de ser una palabra reservada, se muestra esta completa. En caso de ser un *token* variable, se especifica el tipo de *token* (ID, CTE o STR).
- Tercera columna: Lexema. En caso del *token* ser variable, en esta columna se especificará su lexema. En otro caso, estará vacía.

Adicionalmente, de forma intercalada con la identificación de los *tokens*, se mostrarán:

- errores léxicos;
- *warnings* léxicos;
- errores sintácticos; y
- detecciones semánticas relevantes.

Se define como “detección semántica relevante” a la detección de:

- declaraciones de variables;
- declaraciones e invocaciones de funciones;
- expresiones lambda;
- estructuras de control;
- condiciones; y
- programa.

Por último, al final de la salida, es posible observar una tabla de símbolos. En esta, se detallan: el lexema; el tipo, la categoría y el alcance (los cuales serán completados apropiadamente en etapas posteriores); y el número de referencias o apariciones de dicho lexema.


EXA

Experiencia

El análisis léxico fue intuitivo y se implementó con facilidad.

Las dificultades surgieron en el análisis sintáctico. No hubo ninguna complicación en la escritura de las reglas admisibles. No obstante, al momento de escribir las reglas de error, los problemas surgieron. La solución a estos no fue en la gran mayoría de los casos automática, sino que requirió el uso de las herramientas de *debugging* proporcionadas por Byacc/J y un fino análisis del seguimiento. A veces, eso no bastaba: los errores eran pocos intuitivos y la solución era modificar una regla muy distinta de la que anunciaba el error. Sin duda, el mayor tiempo invertido en el proyecto se encuentra en la resolución de errores de la gramática.

A pesar de las complicaciones y las frustraciones generadas, se disfrutó del desarrollo y de los conocimientos adquiridos durante este.

Conclusiones

El diseño de un compilador comprende una constante toma de decisiones que repercutirán directamente en su *performance* y modificabilidad. No es una tarea fácil.

Durante el desarrollo de estas dos primeras etapas, no solo se asentaron los conocimientos vistos en las clases teóricas y se aprendió a utilizar nuevas herramientas, como Byacc/J, sino que también se refrescaron conocimientos de materias anteriores (como Lenguajes Formales y Autómatas y Programación Orientada a Objetos). Adicionalmente, se destaca el trabajo colaborativo como parte fundamental del proyecto, puesto a que fomentó y enriqueció la discusión y surgimiento de las ideas que se ven reflejadas en este.