Trabajo: Presentación Proyecto 2

Presentado por: Alejandro Dávila

21621062

Materia: Teoría de la Computación

Profesor: César Orellana

23 de septiembre de 2018

Contenido

**[Introducción](#_Toc525524482)** [3](#_Toc525524482)

[**Objetivos** 4](#_Toc525524483)

[**1.** **Objetivos específicos** 4](#_Toc525524484)

[**2.** **Objetivo general** 4](#_Toc525524485)

[**Marco teórico** 5](#_Toc525524486)

[**1.** **Autómatas finitos** 5](#_Toc525524487)

[**2.** **Árboles de derivación** 7](#_Toc525524488)

[**3.** **Compiladores** 10](#_Toc525524489)

[**4.** **Antlr** 11](#_Toc525524490)

[**5.** **Autómatas de pila** 12](#_Toc525524491)

[**Contenido** 14](#_Toc525524492)

[**1.** **Diseño de la gramática** 14](#_Toc525524493)

[**2.** **Ejemplos de aplicabilidad** 27](#_Toc525524494)

[**Conclusiones** 33](#_Toc525524495)

[**Bibliografía** 34](#_Toc525524496)

# **Introducción**

Los principios de la computación se remontan al siglo XIXX, cuando Charles Babbage diseñó el motor analítico. Su compañera, Ada Lovelace, diseñó el primer lenguaje de programación de la historia, el cual estaba planificado para el motor analítico. En su honor, posteriormente se nombró un lenguaje de programación (Ada). Sin embargo, este proyecto nunca se pudo poner en práctica, debido a que no existía en aquellos tiempos tecnología ni presupuesto para poder desarrollar el proyecto.

En la primera mitad del siglo XX, se requerían de grandes equipos de personas para hacer cálculos que hoy en día se pueden realizar en periodos de tiempo cortos con una computadora. Estos cálculos eran usados también para la encriptación de datos, usando algunos métodos como el código Morse o encriptación avanzada de comunicaciones, las cuales eran casi imposibles de desencriptar debido a que en el momento en el que se lograba descifrar un algoritmo de encriptación, ya se había desarrollado un algoritmo nuevo. Sin embargo, esto acabó cuando Alan Turing entró en escena, y mediante un arduo estudio matemático, logró elaborar una máquina programable.

Los principios computacionales se basan en reglas matemáticas, y en unidades tan simples y minúsculas como números binarios, que son la base de todos los procesos de un computador. A partir de ellos, se elaboran lenguajes, y estos a su vez forman una gramática.   
El presente informe trata acerca de la gramática computacional, usando como base la gramática de X++, un lenguaje de programación de la plataforma Microsoft Dynamics.

# **Objetivos**

## **Objetivos específicos**

* Demostrar la estructura de una gramática.
* Mostrar las distintas etapas de la compilación de datos.
* Realizar una gramática capaz de interpretar un lenguaje.

## **Objetivo general**

Trabajar en el desarrollo y la comprensión de la aplicación de las gramáticas en el desarrollo de los lenguajes de programación.

# **Marco teórico**

## **Autómatas finitos**

Los autómatas finitos son modelos matemáticos que evalúan cadenas de caracteres. Funcionan como máquinas de estado, o grafos, donde cada carácter se va evaluando hasta que se verifique que se cumple con la condición dada.

Los autómatas finitos se muestran como una tupla, que se interpreta así:

M = <E, A, δ, S, F>

E: conjunto finito de estados

A: alfabeto

δ: transición de estados

* δ: E x A → E si el autómata es determinístico
* δ: E x A → P(E) si el autómata es no determinístico (P(E) es el conjunto potencia de E, es decir el conjunto de todos los subconjuntos de E)

S: estado inicial

F: estado final

Existen dos tipos de autómatas:

* Autómatas finitos determinísticos (DFA)
* Autómatas finitos no determinísticos (NFA)

Los autómatas DFA son aquellos en los que cada estado puede hacer únicamente una transición a otro estado con uno de los símbolos del alfabeto. Estos son los autómatas base, ya que los NFA se evalúan sobre autómatas DFA. En el caso de los DFA, se limitan los caminos posibles de una cadena, ya que cada símbolo solo puede tomar un camino a la vez.

Los autómatas NFA, en cambio, si aceptan dos transiciones distintas para un símbolo del alfabeto. Esto propicia que haya muchas maneras de encontrar el estado final del autómata. Al encontrarnos con un autómata NFA, debemos saber que casi siempre tiene un autómata DFA equivalente, caso contrario al DFA.

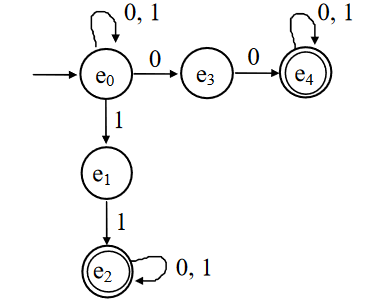
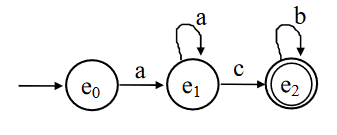
Los autómatas generalmente se expresan por medio de grafos dirigidos, llamados diagramas de estado. También se pueden representar por medio de tablas, donde se enlaza cada símbolo del alfabeto de acuerdo al estado al que se dirige. Cada estado se representa como un nodo en el grafo, y se representan los símbolos de salida de cada estado por medio de flechas que se dirigen a su estado de destino.

Por ende, en un grafo de un autómata DFA solo podemos encontrar una flecha dirigida a otro grafo o a sí mismo por cada símbolo. En los grafos de los autómatas NFA, en cambio, podemos encontrar que hay múltiples flechas con el mismo símbolo dirigiéndose a múltiples destinos.

Los autómatas NFA tienen una variante llamada NFA-Epsilon, en la cual también son incluidas cadenas vacías. Estos autómatas facilitan el proceso de conversión de NFA a DFA, y se usa un proceso de reducción llamado Cerradura Epsilon, con el cual se recortan los caminos del autómata que llevan una cadena vacía.

Ejemplo de un grafo de autómatas:

NFA DFA



(Universidad Nacional del Centro de la Provincia de Buenos Aires, 2009)

## **Árboles de derivación**

Los árboles de derivación son generados a partir de una gramática. De estos árboles se van desprendiendo los caracteres de una cadena por medio de cada una de las producciones gramaticales.

Hay dos tipos de gramáticas:

* Gramáticas libres de contexto
* Gramáticas regulares

Las gramáticas regulares pueden pertenecer al grupo de gramáticas libres de contexto. Sin embargo, no todas las gramáticas libres de contexto pueden ser regulares.

Las gramáticas regulares cumplen con ciertas reglas. Por ejemplo, una gramática regular solo puede estar alineada a una dirección. La definimos a la izquierda o a la derecha. Una gramática libre de contexto, en cambio, puede estar alineada en múltiples direcciones. Una gramática regular debe contener al menos un símbolo terminal por producción, mientras que las gramáticas libres de contexto pueden no tener símbolos terminales, pero pueden tener, en cambio, múltiples símbolos no terminales.

En un árbol de derivación, cada nodo va a tener un símbolo, sea terminal o no terminal. Del nodo se raíz se parte al resto de producciones, empezando con el nodo S. Para elaborar el árbol, debemos tener una cadena a evaluar. Empezamos con el primer carácter de la cadena, revisamos con qué producción cumple, y así seguimos derivando el árbol. Entonces, por cada nodo que busquemos una producción, se generan nuevos nodos hijos, y este proceso se repite sucesivamente hasta que le demos fin a la cadena.

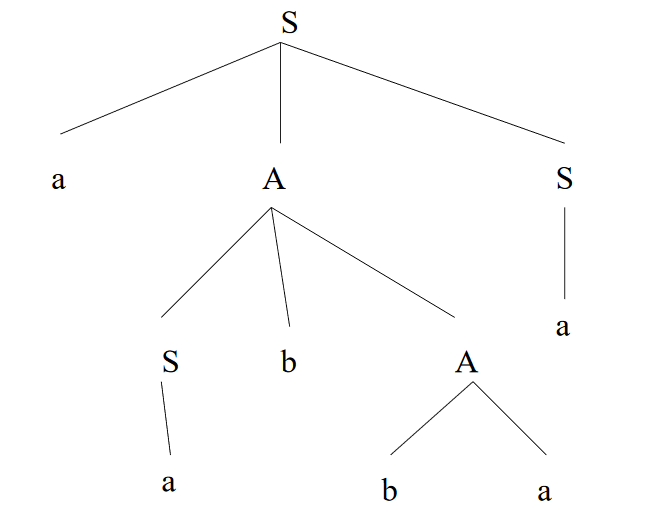
Para leer la palabra generada, se debe leer el árbol de izquierda a derecha.

Un ejemplo de un árbol de derivación es el siguiente:

Tenemos la gramática S, que se compone de las siguientes producciones:

S -> aAS | a | SbA | SS | ba

Como podemos ver, esta gramática es libre de contexto, por lo cual el árbol se derivará en varias direcciones. A continuación, se evaluará la cadena *aabbaa*.



Como podemos observar, este árbol no está derivado a un solo lado. Sin embargo, si lo leemos de izquierda a derecha, podemos observar que la cadena *aabbaa* si se cumple. Por ende, podemos considerar que esta cadena forma parte del lenguaje.

Cuando una cadena puede generar la misma cadena en árboles distintos, se le considera a la gramática una gramática ambigua, y si toda la gramática que genera es ambigua, al lenguaje también se le considera ambiguo. Una gramática deja de ser ambigua si le eliminamos una producción.

Luego, una gramática muchas veces cuenta con símbolos inútiles. Un símbolo es útil solo si puede generar una derivación, y una producción es útil si todos sus símbolos son útiles. Esto significa que todos los símbolos útiles aplican a una palabra asociada al lenguaje que genera la gramática.

Si nosotros eliminamos producciones inútiles, esto no cambiará el lenguaje producido por una gramática, y no tiene mayor relevancia.

Para eliminar producciones inútiles, podemos usar el siguiente algoritmo:

1. Eliminar las variables inalcanzables en todas las producciones donde aparezcan.
2. Eliminar símbolos no terminales que no sean alcanzables desde S, y todas sus producciones.

Para aplicar estos algoritmos, es muy importante hacerlo en el orden indicado, ya que, si aplicamos primero el segundo paso, básicamente no estamos borrando nada.

(Universidad de Granada, 2018)

Una forma de eliminar producciones y símbolos inútiles es aplicando la forma normal de Chomsky.

La definición formal de Chomsky es que la gramática cumpla con lo siguiente:

A → BC con A,B,C ∈ N

A →a, con A,B ∈ N y a ∈ T

El teorema de Chomsky es el siguiente:

“Todo lenguaje incontextual L que no incluye la cadena vacía, es generado por una gramática en FNC.”

(Universidad Politécnica de Valencia, 2016)

## **Compiladores**

Un compilador es aquel que traduce el código escrito en lenguaje humano a lenguaje de máquina. Los compiladores tienen cinco fases: análisis léxico, parsing, análisis semántico, optimización, y generación de código.

La primera fase, el análisis léxico, se encarga de separar el código por palabras, llamadas lexemas o tokens. Normalmente, estos tokens son almacenados en tablas, que luego son transformadas a árboles de derivación, de donde se procede a ejecutar el análisis semántico.

Los tokens son divididos de acuerdo a las cadenas o caracteres. Los divide por palabras reservadas, instrucciones, delimitadores, etc.

La siguiente etapa es el parsing. En el parsing es donde se extraen los tokens de las tablas para luego ser evaluados en árboles. Entonces, por ejemplo, las palabras en la tabla que pertenecen a la estructura del if, son integradas en un árbol que divide los tokens de acuerdo a la estructura del if. Luego el parser va extrayendo las distintas partes del código, formando los distintos árboles y luego integrándolos en un solo árbol común.

Al haber sido ejecutados el análisis léxico y el parsing, sin embargo, aún no ha sido interpretado qué es lo que debe realizar el programa. Es ahí donde entra el análisis semántico. Mediante el análisis semántico, se eliminan las ambigüedades que se encuentran en el código. Por ejemplo, en el análisis léxico y el parsing se puede incluir una palabra que es igual a una palabra reservada, y probablemente pase de estas dos fases. En el análisis semántico, sin embargo, se analizan las diferencias, y se determina que esta palabra no puede ser variable porque es una palabra reservada.

Luego, en la siguiente etapa, la optimización, se busca que el código sea lo más eficiente posible, para de esta manera ser ejecutado sin problemas por la máquina.

(Aiken, 2015)

## **Antlr**

Antlr es una herramienta de parsing en la que se puede leer, procesar, ejecutar, o traducir lenguaje estructurado y archivos binarios. Es ampliamente usado en el mundo de la programación para elaborar lenguajes, así como herramientas adicionales y frameworks.

Muchos softwares famosos, como Twitter, Oracle, y Netbeans, para realizar parsing de múltiples lenguajes de programación. Antlr tiene muchas aplicaciones en la programación, como aplicaciones para leer archivos de texto, convertidores de lenguajes de programación, etc.

De una descripción de lenguaje formal, llamada gramática, Antlr genera un parser para ese lenguaje, que automáticamente genera un árbol de derivación, que se usa para demostrar la manera en la que es construida la gramática.

Antlr es considerada una de las herramientas de parsing más influyentes en el mundo de la programación, siendo ampliamente utilizada para este fin. Antlr ha influido mucho en las siguientes áreas:

* Vista previa linear aproximada
* Predicados sintácticos y semánticos
* Árboles de derivación.

Antlr está disponible para una gran variedad de lenguajes de programación. Es un software de código libre, y está disponible en su portal de GitHub.

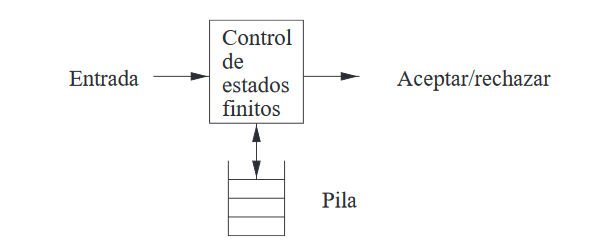
(Parr, 2014)

## **Autómatas de pila**

Los autómatas de pila son utilizados para definir lenguajes de libre contexto. En términos simples, un autómata de pila es básicamente un autómata no determinístico Epsilon (NFA-E), con la diferencia que se le agrega una pila para evaluar las cadenas.

La presencia de una pila en el autómata de pila genera una diferencia muy marcada con los NFA-E, y es que los autómatas de pila cuentan con un espacio infinito. Esto le permite al autómata de pila poder almacenar una cantidad infinita de símbolos. Sin embargo, a diferencia de las computadoras generales, que tienen un espacio finito, estas pueden almacenar cantidades arbitrarias de información, y acceder a ella de manera arbitraria, mientras que los autómatas de pila solo pueden acceder al primer elemento en la pila, haciendo pop del primer elemento que encuentra.

Este hecho limita mucho la funcionalidad de un autómata de pila. La máquina de Turing es similar al autómata de pila, con la gran diferencia que la máquina de Turing puede moverse de izquierda a derecha para acceder a la información y escribir en los distintos espacios de memoria.



El autómata de pila funciona de una manera similar a la que está demostrada en el diagrama. Un control de estados finitos lee las entradas una a una, y lleva a cabo su transición de acuerdo al estado actual, el símbolo que viene entrando y el símbolo que está de primero en la pila. También puede hacer transiciones espontáneas, que usan un símbolo Epsilon.

El autómata de pila realiza lo siguiente:

1. Consume de la entrada el símbolo que usa en la transición. Si como entrada se utiliza ε, entonces no se consume ningún símbolo de entrada.
2. Pasa a un nuevo estado, que puede o no ser el mismo que el estado anterior.
3. Reemplaza el símbolo de la parte superior de la pila por cualquier cadena. La cadena puede ser ε , lo que corresponde a una extracción de la pila. Podría ser el mismo símbolo que estaba anteriormente en la cima de la pila; es decir, no se realiza ningún cambio en la pila. También podría reemplazar el símbolo de la cima de la pila por otro símbolo, lo que cambiaría la cima de la pila, pero no añade ni extrae ningún símbolo. Por último, el símbolo de la cima de la pila podría ser reemplazado por dos o más símbolos, lo que (posiblemente) tendría el efecto de cambiar el símbolo de la cima de la pila, añadiendo después uno o más nuevos símbolos a la pila.

(Hopcroft, Motwani, & Ullman, 2008)

# **Contenido**

## **Diseño de la gramática**

La gramática demostrada a continuación fue diseñada mediante la herramienta de parsing Antlr, desarrollada en el lenguaje de programación Java mediante el IDE IntelliJ de JetBrains.

Esta gramática se elaboró a partir de samples de código del lenguaje de programación X++. Este lenguaje de programación forma parte del entorno de desarrollo Microsoft Dynamics, y se distingue particularmente por su fusión entre lenguajes de programación derivados de C y SQL.

Es importante identificar primero la estructura de este lenguaje. Al ser una fusión de C++ y SQL, cuenta con una gramática muy amplia, debido a que contiene muchos elementos en común con ambos lenguajes.

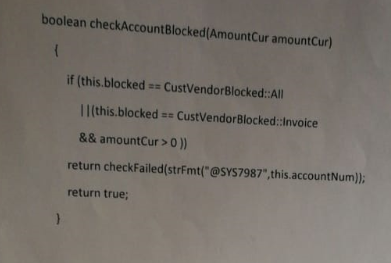
En general, X++ cuenta con los siguientes tipos de datos:

* AnyType: un espacio de memoria en el cual es asignado una variable. Al ser un espacio de memoria, soporta el resto de tipo de datos. Es el equivalente al var de C#.
* Boolean: tipo de dato booleano cuyo valor asignado solo puede ser true o false. Su palabra reservada es boolean.
* Dates: Contiene día, mes, y año.
* Integer: un número sin valor decimal. Su palabra reservada es int.
* Reals: un número con valor decimal, también llamado decimal. Es el equivalente a float o double.
* Strings: cadenas de caracteres. Están representados por la palabra reservada str.
* TimeOfDay: dato que contiene horas, minutos y segundos. Su palabra reservada es timeOfDay.
* UtcDateTime: dato que contiene año, mes, día, hora, minutos, y segundos.
* Void: dato que no devuelve ningún valor.

X++ cuenta con las siguientes palabras reservadas:

|  |  |
| --- | --- |
| Tipo de token | Palabra reservada |
| T\_Whitespace | \t, \r, \n |
| T\_numbers | [0-9] |
| T\_strings | [a-z] |
| T\_ops\_booleans | && |  | | |  == |  < |  >|  <= |  >= |
| T\_boolean\_types | true |  false |
| T\_boolean | boolean |
| T\_string | str |
| T\_int | int |
| T\_date | date |
| T\_time\_of | timeOfDay |
| T\_reals | decimals |
| T\_utc | utcDateTime |
| T\_anyT | anyType |
| T\_void | void |
| T\_comillas | “” |
| T\_comma | , |
| T\_delim  T\_operator | ;  + |  - |  \* |  / | |
| T\_equals | = |
| T\_this | this |
| T\_dot | . |
| T\_static\_caller | :: |
| T\_open\_par | ( |
| T\_closing\_par | ) |
| T\_if | if |
| T\_else | else |
| T\_return | return |
| T\_static\_access | static |
| T\_while | while |
| T\_select | select |
| T\_index | index |
| T\_hint | hint |
| T\_selAll | \* |
| T\_from | from |
| T\_where | where |
| T\_count | ++ |
| T\_opening\_b | { |
| T\_closing\_b | } |

A continuación, se presentan los sample codes a partir de los cuales se elaboraron las gramáticas.

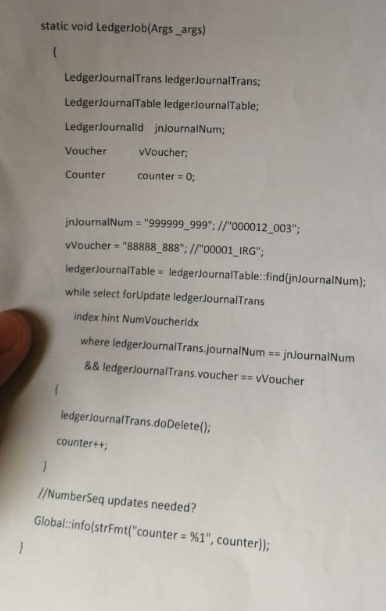


La primera función, checkAccount, es una función que devuelve un valor boolean. Como podemos observar, su estructura es bastante simple. Cuenta con un bloque If, que como podemos observar, en caso de solo tener una instrucción después de la condición, no necesita usar brackets para cerrar el enunciado, sino que admite la forma

if (condición) instrucción;

En este caso, la condición del if incluye un comparador anidado booleano. Luego, los valores de retorno utilizan también comparadores booleanos, siendo el que prosigue la condición del if una función, y en su defecto, devuelve true.

Asimismo, podemos observar que en X++ se pueden llamar los atributos de manera similar a los lenguajes derivados de C. Por medio de un this. podemos acceder a los atributos locales de la clase, y por medio del uso del doble doble punto (::), podemos acceder a las funciones y atributos estáticos (static) de otras clases. Asimismo, la función acepta parámetros de distintas clases.

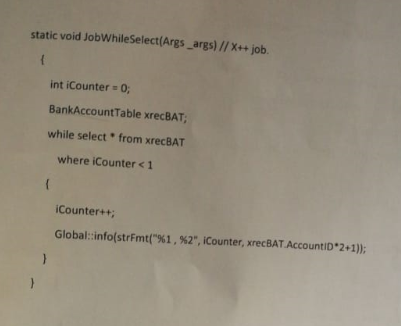


La segunda función, LedgerJob, es una función void, lo que significa que no devuelve ningún valor. Es importante notar que tiene tipo de acceso static, lo que significa que se puede acceder a esta función sin necesidad de crear un objeto de su clase. En los parámetros podemos ver que recibe Args, que por lo general en los lenguajes derivados de C simboliza texto.

En esta función podemos ver como se declaran los atributos en X++. Funcionan de la misma manera que en casi todos los lenguajes de programación, pudiendo asignar valores constantes, valores variables, y valores devueltos por funciones.

En el while, podemos ver que difiere mucho de la estructura normal del ciclo while en los lenguajes derivados de C. En lugar de escribirse en el patrón while (condición) instrucción, el while de X++ utiliza como condición una sentencia SQL.

Podemos ver que los contadores funcionan igual que en la mayoría de los lenguajes de programación, y las funciones soportan varios parámetros.



En la tercera función, podemos ver que se declara de la misma manera que la segunda, un static void. También recibe como parámetro args, y contiene un código similar al de la segunda función, utilizando declaraciones, contadores, llamados estáticos, y un while controlado por una sentencia SQL.

Es importante identificar la manera en la que se escribe el código en X++, ya que a partir de un análisis de la estructura y de las palabras que se utilizan es como se puedo construir una gramática que emule la manera en la que está declarada la gramática estándar del lenguaje X++. Es reconocible su similitud con los lenguajes derivados de C, al usar brackets como delimitadores, punto y coma, y otro sinfín de características que nos hacen recordar el código de los lenguajes derivados de C.

La creación de una gramática se compone de dos tipos de elementos:

* Tokens
* Parsers

Los tokens son aquellos caracteres que se utilizan para la gramática. Un token realiza la función de reservar una palabra para la gramática, de modo que esta palabra no podrá ser utilizada para otras funciones. Por ejemplo, si this es declarado como un token, aunque sea llamado como string estándar no funcionará, debido a que al ser declarado token se convierte en una palabra reservada. Los títulos de los tokens se declaran en letras mayúsculas, y sus valores son declarados entre comillas simples.

Los parsers son aquellos que se encargan de procesar la gramática. En los parsers, nosotros le decimos a la gramática como se deben organizar las cadenas de código de nuestro lenguaje. Un parser puede acceder a una múltiple cantidad de datos, así como a otros parsers dentro de la misma clase. Los parsers funcionan de igual manera que una gramática. Podes usar recursividad en los parsers, que se puede usar sin restricciones, ya que la gramática que se declara en Antlr es al gusto del desarrollador, pudiendo ser regular o libre de contexto. Los parsers se declaran en minúsculas.

Si bien es una buena práctica acorde a los estándares dividir la información entre tokens y parsers, es posible obviar el uso de tokens y limitarse a usar únicamente los parsers, debido a que estos soportan la declaración de valores. Sin embargo, por cuestiones de compatibilidad, legibilidad, y orden, es importante separar la información entre tokens y parsers, lo cual nos provee un mejor control de nuestra gramática, y nos ayuda a ahorrar recursos de memoria.

Para elaborar una gramática, es recomendable empezar de abajo para arriba. Por cuestiones de estandarización, los tokens son escritos en la parte de abajo del archivo, y los parsers arriba.

Lo primero que se debe hacer es escribir los tokens para saber de dónde partir al momento de escribir los parsers, y de esta manera poder usarlos desde que los primeros parsers son escritos.

Es importante clasificar los tokens de acuerdo al rol que desempeñan dentro de la gramática. Si bien if y else forman parte de la estructura zvaluativa del código, es importante colocar ambos en un token separado, ya que, si los colocamos a ambos en un token de evaluación, podríamos usar el else como un if, que al llegar al analizador semántico detectaría un error.

Es recomendable colocar la mayoría de las palabras reservadas como tokens separados. De esta manera, evitamos confusiones al momento de elaborar la gramática, y le ahorramos trabajo al analizador semántico.

Identificar las palabras reservadas es clave para una elaboración exitosa de la gramática, ya que en ocasiones se vuelve común confundir nombres de clases creadas por el programador como palabras reservadas. Otra manera de evitar confusiones con los strings es separar los distintos tipos de palabras, aunque usen la misma estructura. Por ejemplo, si bien las variables y las clases son escritas de la misma manera, es importante hacer énfasis en ambos, ya que al momento de ser interpretado por un analizador cumplen con funciones totalmente distintas.

Al igual que los números y los strings, todos los demás caracteres que estén representados en la gramática deben estar representados como tokens, e identificados de acuerdo a su función. Por ejemplo, no se pueden agrupar ambos paréntesis en un token, porque se escriben de manera distinta en el código, y cumplen con funciones distintas,

Al terminar con la declaración de los distintos tokens, se debe pasar a la declaración de parsers. Los parsers, que son escritos en letras minúsculas, cumplen con la función de interpretar las cadenas que son ingresadas y evaluar si son compatibles con la gramática del lenguaje.

Los parsers pueden estar compuestos por tokens y por otros parsers. Los tokens funcionan como terminales, mientras que cada parser funciona como no terminal. Es importante reconocer que no todo debe estar agrupado en un solo parser, debido a que esto hace el código ilegible. Se debe separar los parsers en la cantidad que sea necesaria, y clasificarlos de acuerdo a la necesidad de la gramática.

Al igual que en una gramática, los parsers pueden aplicar recursividad. De esta manera, podemos evitar el código repetitivo, y hacemos el parser más eficiente. Al poder usar la recursividad y el acceso a otros parsers, podemos hacer combinaciones que efectúan una buena gramática, y que podemos entender con facilidad.

Los parsers unen las cadenas que son ingresadas. Las va separando de acuerdo al token, y dependiendo del parser que esté invocando, crea un nuevo nodo con hijos. El árbol de derivación que genera Antlr es fácil de entender, y clasifica todos los tokens de acuerdo a la manera que declaramos en el parser.

Es importante recalcar que se debe seguir un orden al momento de declarar los tokens y los parsers. Muchas veces Antlr reporta errores debido a que un token necesario no es evaluado por estar debajo de otro token, etc. Dicho problema se soluciona escribiendo los tokens de abajo para arriba, y escribiendo los parsers en un orden lógico. Asimismo, se debe manejar con sumo cuidado el uso de símbolos o caracteres especiales, ya que estos están reservados por Antlr, y deben ser declarados de la manera correcta para no provocarnos errores.

A continuación, se muestra el código de muestra de la gramática de X++:

|  |
| --- |
| grammar Declaraciones; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| /\* |
|  |

|  |
| --- |
| Parser |
|  |

|  |
| --- |
| \*/ |
|  |

|  |
| --- |
| string: (' "' STR '" ')\*; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| anytype: BOOL | string | NUMBER | STR; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| var : STR; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| values: NUMBER | var; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| arithmeticOperation : values OPERATOR values arithmeticOperation |
|  |

|  |
| --- |
| | values OPERATOR values; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| toEqual: anytype | var | arithmeticOperation; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| declare: data\_types var EQUALS toEqual DELIMITATOR declare |
|  |

|  |
| --- |
| | data\_types var EQUALS toEqual DELIMITATOR |
|  |

|  |
| --- |
| | data\_types var DELIMITATOR declare |
|  |

|  |
| --- |
| | data\_types var DELIMITATOR |
|  |

|  |
| --- |
| | data\_types var EQUALS callFunc DELIMITATOR declare |
|  |

|  |
| --- |
| | data\_types var EQUALS callFunc DELIMITATOR |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| data\_types: BOOLEAN |
|  |

|  |
| --- |
| | STRING |
|  |

|  |
| --- |
| | INTEGER |
|  |

|  |
| --- |
| | DATE |
|  |

|  |
| --- |
| | TIMEOF |
|  |

|  |
| --- |
| | REALS |
|  |

|  |
| --- |
| | UTC |
|  |

|  |
| --- |
| | ANYTYPE |
|  |

|  |
| --- |
| | VOID; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if: IF evaluator return DELIMITATOR |
|  |

|  |
| --- |
| | IF evaluator return return DELIMITATOR |
|  |

|  |
| --- |
| | IF evaluator return DELIMITATOR ELSE return DELIMITATOR; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| evaluator: OPENPAR conditions CLOSEPAR |
|  |

|  |
| --- |
| | OPENPAR conditions BOOL\_OPS OPENPAR conditions CLOSEPAR CLOSEPAR |
|  |

|  |
| --- |
| | conditions BOOL\_OPS conditions |
|  |

|  |
| --- |
| | conditions; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| conditions: caller BOOL\_OPS caller |
|  |

|  |
| --- |
| | caller BOOL\_OPS caller BOOL\_OPS caller BOOL\_OPS caller; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| caller: THIS DOT var | className STATIC\_CALLER var | anytype; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| return: RETURN anytype | RETURN callFunc; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| className: STR; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| forSelect: SELALL | STR | var className INDEX HINT className ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| select: SELECT forSelect FROM className WHERE evaluator | SELECT forSelect WHERE evaluator; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| coma: ', '; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| callFunc: var DOT var |
|  |

|  |
| --- |
| | THIS DOT var |
|  |

|  |
| --- |
| | callFunc OPENPAR CLOSEPAR |
|  |

|  |
| --- |
| | var OPENPAR callFunc CLOSEPAR |
|  |

|  |
| --- |
| | var STATIC\_CALLER var OPENPAR var OPENPAR (string coma)\* (string coma)\* var (coma)\* callFunc SELALL arithmeticOperation CLOSEPAR |
|  |

|  |
| --- |
| | var OPENPAR (string coma)\* callFunc CLOSEPAR |
|  |

|  |
| --- |
| | var callFunc |
|  |

|  |
| --- |
| | var STATIC\_CALLER var OPENPAR var CLOSEPAR |
|  |

|  |
| --- |
| | var COUNT; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| whileins: callFunc DELIMITATOR whileins |
|  |

|  |
| --- |
| | callFunc DELIMITATOR |
|  |

|  |
| --- |
| ; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| while: WHILE (select '{ ')\* (whileins ' }')\*; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| funcHeader: data\_types var OPENPAR className var CLOSEPAR |
|  |

|  |
| --- |
| | STATIC\_ACCESS funcHeader; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| function: (funcHeader '{ ')\* if (' }')\* |
|  |

|  |
| --- |
| | (funcHeader '{ ')\* declare while callFunc (' }')\* |
|  |

|  |
| --- |
| | (funcHeader '{ ')\* declare while (' }')\*; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| /\* |
|  |

|  |
| --- |
| Lex |
|  |

|  |
| --- |
| \*/ |
|  |

|  |
| --- |
| COUNT: '++'; |
|  |

|  |
| --- |
| WHERE: 'where'; |
|  |

|  |
| --- |
| FROM: 'from'; |
|  |

|  |
| --- |
| SELALL: '\*'; |
|  |

|  |
| --- |
| HINT: 'hint'; |
|  |

|  |
| --- |
| INDEX: 'index'; |
|  |

|  |
| --- |
| SELECT: 'select'; |
|  |

|  |
| --- |
| WHILE: 'while'; |
|  |

|  |
| --- |
| STATIC\_ACCESS: 'static'; |
|  |

|  |
| --- |
| RETURN: 'return'; |
|  |

|  |
| --- |
| ELSE: 'else'; |
|  |

|  |
| --- |
| IF: 'if'; |
|  |

|  |
| --- |
| CLOSEPAR: ')'; |
|  |

|  |
| --- |
| OPENPAR: '('; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| STATIC\_CALLER: '::'; |
|  |

|  |
| --- |
| DOT: '.'; |
|  |

|  |
| --- |
| THIS: 'this'; |
|  |

|  |
| --- |
| DELIMITATOR: ';'; |
|  |

|  |
| --- |
| EQUALS: '='; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| OPERATOR : '+' |
|  |

|  |
| --- |
| |'\*' |
|  |

|  |
| --- |
| |'/' |
|  |

|  |
| --- |
| |'-'; |
|  |

|  |
| --- |
| CLOSECOMI: '"'; |
|  |

|  |
| --- |
| OPENCOMI: '"'; |
|  |

|  |
| --- |
| VOID: 'void'; |
|  |

|  |
| --- |
| ANYTYPE: 'anyType'; |
|  |

|  |
| --- |
| UTC: 'utcDateTime'; |
|  |

|  |
| --- |
| REALS: 'decimal'; |
|  |

|  |
| --- |
| TIMEOF: 'timeOfDay'; |
|  |

|  |
| --- |
| DATE: 'date'; |
|  |

|  |
| --- |
| INTEGER: 'int'; |
|  |

|  |
| --- |
| STRING: 'str'; |
|  |

|  |
| --- |
| BOOLEAN : 'boolean'; |
|  |

|  |
| --- |
| BOOL: 'true' | 'false'; |
|  |

|  |
| --- |
| BOOL\_OPS: ' && ' |
|  |

|  |
| --- |
| | '||' |
|  |

|  |
| --- |
| | '==' |
|  |

|  |
| --- |
| | '<' |
|  |

|  |
| --- |
| | '>' |
|  |

|  |
| --- |
| | '<=' |
|  |

|  |
| --- |
| | '>='; |
|  |

|  |
| --- |
| STR: [[a-zA-Z]+ ; |
|  |

NUMBER: [0-9]+ ;

|  |
| --- |
| WHITESPACE : [ \t\r\n]+ -> skip; |

Como se puede observar, en las primeras líneas de código podemos encontrar los parsers. Dichos parsers anidan otros parsers en algunos casos, para simbolizar la estructura de anidación de datos que tienen las funciones en general.

Es importante recalcar que algunos tokens deben ser almacenados en un parser antes de ser utilizados. Esto porque el llamar dos tokens seguidos puede provocar errores de concatenación, por lo que es necesario almacenar los tokens en parsers.

Debajo de los parsers, podemos apreciar los tokens. Dichos tokens están ordenados en el mayor orden lógico posible para evitar errores de parser. Es importante tener de primeros los valores strings y numéricos, ya que los parsers dependen a gran escala de los valores string y numéricos.

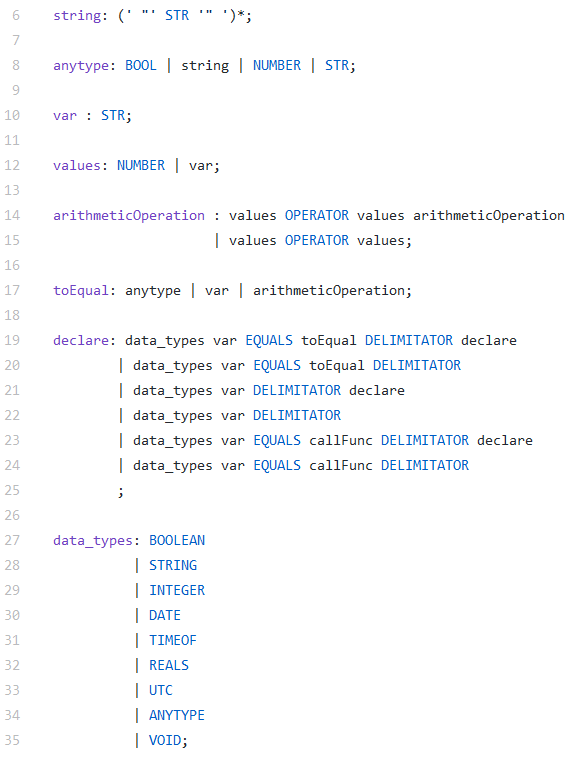
## **Ejemplos de aplicabilidad**

**Declaración de variables y operaciones matemáticas**

X++ utiliza una declaración de variables muy similar a la de otros lenguajes de programación. Estas variables pueden ser de los tipos anteriormente especificados, y se pueden declarar de maneras variadas, ya sea igualándolas a operaciones matemáticas, valores, u otras variables.

Las variables son de tipo string. Van precedidas por el tipo de dato, y están acompañadas en su declaración por las condiciones anteriormente expuestas, así como también pueden ser declaradas sin valor asignado.

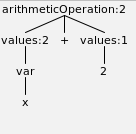
El parser de declaración de variables, por cumplir con varias condiciones, depende de otros parsers, así como se muestra en la imagen a continuación.

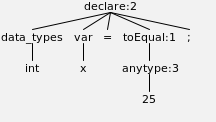
Como se puede observar, el parser base para declarar variables es declare. Este tiene la capacidad de asignar valores y ejecutar operaciones. Recibe su tipo de dato de un parser que a su vez recibe los tokens de tipos de datos, y utiliza un equal, y el parser toEqual que funciona como la instrucción ejecutada por el parser declare a partir de él.

El parser declare funciona de manera recursiva cuando necesitamos declarar múltiples variables en una función.

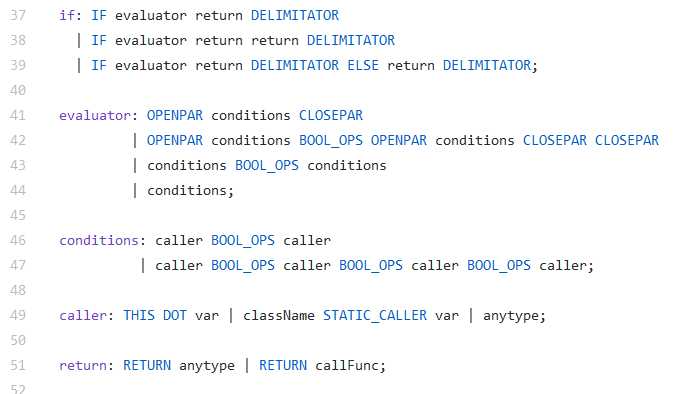
Otro parser que realiza operaciones es arithmeticOperations, el cual recibe dos parámetros y un operador para realizar las operaciones correspondientes.

Ejemplos de árboles de derivación:

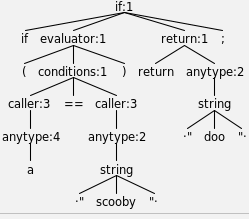




**If**

El if de X++ funciona similar al de C. Este utiliza parsers de condiciones y evaluadores, los cuales se anidan en el parser if, que es el encargado de recolectar los demás parsers y convertirlos en uno solo.

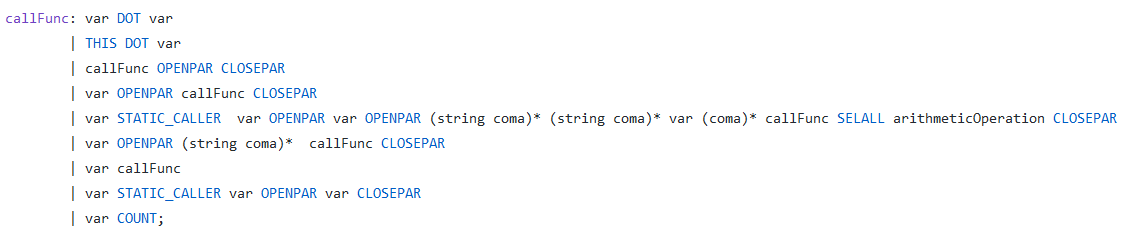
Ejemplo de árbol de derivación:



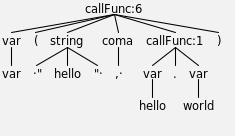
**Funciones**

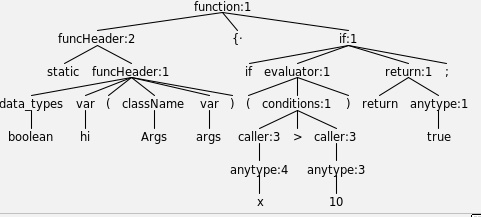
Debemos empezar poniendo especial atención a la declaración de funciones. Debido a la complejidad que pueden suponer algunas funciones, este parser depende mucho en la recursión, ya que debe economizar en la mayor cantidad posible de trabajo la declaración de funciones, y brinda una mayor flexibilidad al momento de escribir funciones complejas.

Por último, la base que engloba todos los parsers anteriormente expuestos son las functions. Este es el parser con el que más cuidado se debe tener, debido a que por la cantidad de tokens que tiene, es muy complicado poder hacer un debugging efectivo, y nos dificulta la tarea de evaluar errores.



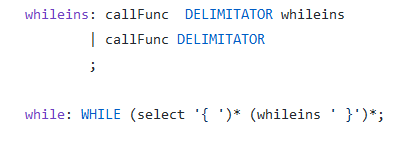
Ejemplo de árbol de derivación:



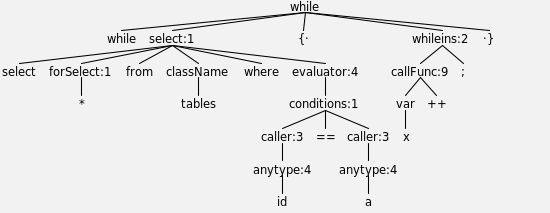


**While**

Los ciclos while, a diferencia de la mayoría de los lenguajes, en X++ funcionan a partir de sentencias select. Como pudimos observar en los tokens, la mayoría de las funciones select ya están incluidas en el lenguaje.



Ejemplo de árbol de derivación:



# **Conclusiones**

* Es útil hacer gramáticas para poder evaluar cómo se desarrollan los lenguajes de programación.
* A partir de un lenguaje se construyen gramáticas que pueden suplir sus necesidades.
* Los árboles de derivación son la forma más efectiva de probar el desarrollo de un autómata.

# **Bibliografía**

Aiken, A. (2015). *01-02: Structure of a Compiler.* Obtenido de Stanford University - Self-Paced Compiler: https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/courseware/ed92b25157d94fbe9874370673ca0a58/2e3841013edf40feb9ba173c7c10225e/

Hopcroft, J., Motwani, R., & Ullman, J. (2008). *Introducción a la teoría de autómatas, lenguajes, y computación.* Madrid: Pearson.

Parr, T. (2014). *About Antlr.* Obtenido de Antlr.org: https://www.antlr.org/about.html

Universidad de Granada. (2018). *Publicaciones Docencia.* Obtenido de Universidad de Granada: ftp://decsai.ugr.es/pub/utai/other/smc/docencia/tr4.pdf

Universidad Nacional del Centro de la Provincia de Buenos Aires. (2009). *Autómatas finitos.* Obtenido de http://www.exa.unicen.edu.ar/catedras/ccomp1/ApunteAutomatasFinitos.pdf

Universidad Politécnica de Valencia. (2016). *Práctica 6 - Forma Normal de Chomsky.* Obtenido de Universidad Politécnica de Valencia: http://users.dsic.upv.es/asignaturas/facultad/tal/practica6.pdf