Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Organización de lenguajes y compiladores 2
Primer Semestre 2016
Catedráticos: Ing. Bayron López, Ing. Edgar Sabán
Tutores académicos: Julio Flores, Andrea Alvarez, Jordy González, Kevin Lajpop

IDE - SINTAX

Primer proyecto de laboratorio

1 OBJETIVOS

1.1 OBJETIVO GENERAL

1. Que el estudiante sea capaz de construir una solución de software con la capacidad de generar analizadores sintácticos ascendentes mediante una herramienta de análisis descendente.

1.2 OBJETIVOS ESPECÍFICOS

- Que el estudiante implemente el uso de atributos sintetizados y heredados por medio de traducción dirigida por la sintaxis visto en clase.
- Que el estudiante integre el analizador léxico generado en la primera fase del proyecto para realizar el análisis de sintaxis
- Que el estudiante integre los conocimientos adquiridos en el curso prerrequisito de Compiladores 1 con el curso actual.

2 DESCRIPCIÓN GENERAL DEL PROYECTO

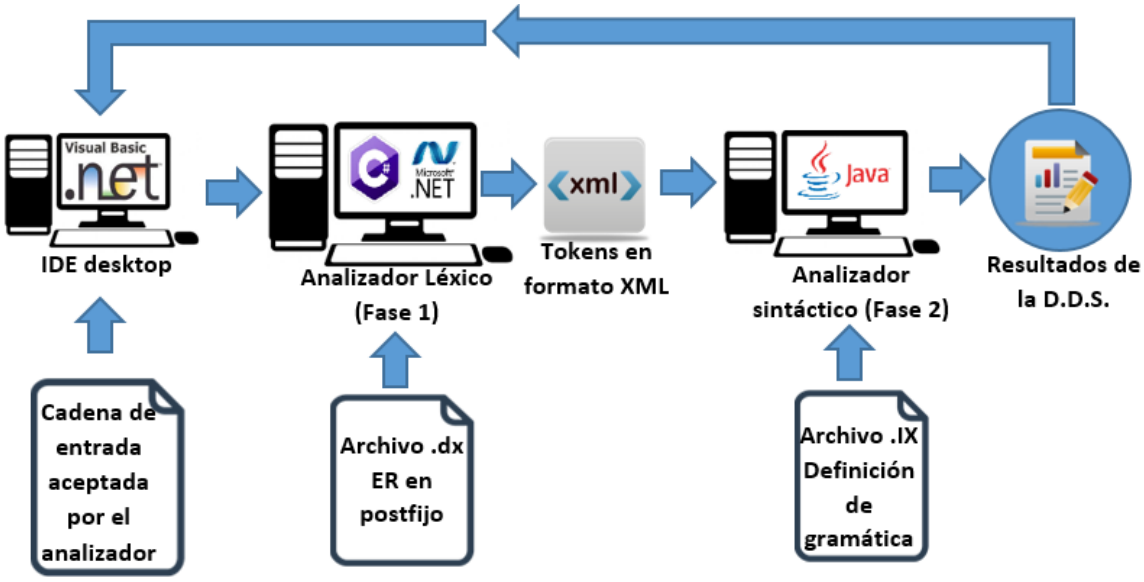
En esta nueva fase de proyecto, mediante una herramienta de análisis sintáctico descendente (JavaCC), se debe crear una herramienta de análisis sintáctico ascendente que funcione con el analizador LR(1) y pueda integrarse al analizador léxico creado en la práctica y a un IDE desde el cual debe llamarse la ejecución de todo el proceso.

La mayoría de herramientas de análisis sintáctico funcionan de manera que su análisis va de los hijos a la raíz (ascendente) y utilizan el algoritmo LALR(1) para hacerlo, esto abarca un gran número de gramáticas que pueden ser aceptadas. Un análisis descendente analiza de la raíz hacia las hojas y requiere de un algoritmo LL(K) lo cual restringe la gramática ya que existen ciertas reglas para que nuestra gramática sea aceptada (eliminación recursividad por la izquierda, factorización y no ambigüedad).

La herramienta por crearse debe ejecutar definiciones dirigidas por la sintaxis o esquemas de traducción. En dichas definiciones se puede trabajar con atributos heredados y sintetizados tal como se enseña en la clase magistral.

El IDE que se plantea para el uso de las herramientas de análisis tiene como objetivo el mandar solicitudes de parsing y visualizar el resultado de las acciones incrustadas en la gramática. Así mismo debe generar un archivo de texto plano donde se muestran los resultados de las acciones para su uso posterior (segundo proyecto), creando así una herramienta funcional capaz de realizar análisis léxico y sintáctico hecho por el estudiante.

Flujo de la aplicación a realizar:



El IDE se comunica con las herramientas de análisis léxico y sintáctico creadas utilizando sockets para enviar solicitudes de analizar código fuente de alto nivel. Las herramientas de análisis creadas se comunican entre ellas para transferir el listado de tokens vía Apache Thrift. Por último el resultado de la DDS (definición dirigida por la sintaxis) o EDT (esquema de traducción) es mostrado al IDE que hizo la solicitud de analizar código fuente, dicho IDE tiene la posibilidad de generar un archivo de texto plano donde se muestre el resultado de la DDS o EDT (tomar en cuenta que dicho archivo será de importancia en el proyecto final, ya que será la forma en que se generará código de 3 direcciones por lo que será un requerimiento obligatorio).

3 ARQUITECTURA DE LA APLICACIÓN

La arquitectura de la aplicación se compone de 3 módulos que harán funcionar a la aplicación como un solo programa:

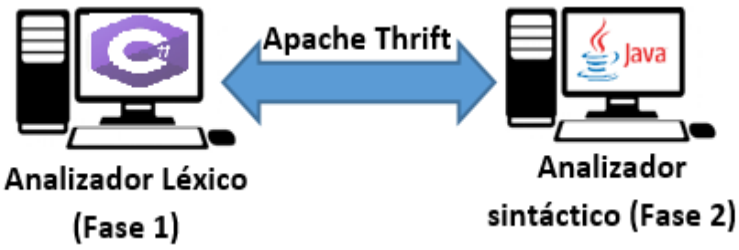
1. Integración con el analizador léxico de la fase 1 (práctica)
2. Generador de analizadores sintácticos
3. Integración de ambos analizadores con el IDE

Los 3 componentes de la arquitectura se describen a continuación:

3.1 INTEGRACIÓN CON LA PRÁCTICA (FASE 1)

En la práctica (fase 1 del proyecto) se realizó un generador de analizadores léxicos el cual tomaba una serie de expresiones regulares escritas en postfijo para crear un analizador y al analizar generaba un listado de tokens en un formato dado (formato XML). Dicho listado de tokens será utilizado en esta parte del proyecto para hacer el proceso de parsing. La comunicación entre el analizador léxico y sintáctico se realizará mediante la herramienta de comunicación de lenguajes Apache Thrift.

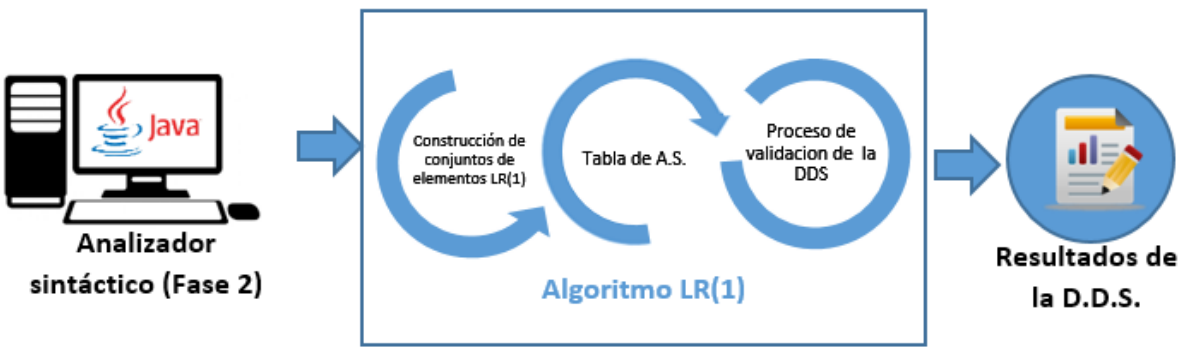
Arquitectura física:



Con el editor de archivos .dx (creado en la fase anterior) se ingresará las expresiones regulares y reglas léxicas después se deberá cargar a memoria el analizador léxico y el programa quedará en modo de escucha donde estará disponible para recibir solicitudes de análisis léxico. Cuando el analizador léxico recibe un código proveniente de un IDE este analizará y generará el listado de tokens. Después automáticamente pasa el listado de tokens al analizador sintáctico que de igual forma estará cargado a memoria con un estado de escucha y se analizará el listado de tokens. Por último el analizador genera el resultado de las acciones de la gramática y los comunica al IDE que solicitó el análisis (no es necesario que el programa soporte concurrencia).

3.2 GENERADOR DE ANALIZADORES DE SINTAXIS

Para el generador de análisis sintaxis se construirá un analizador LR(1), para lo cual deberá generar una tabla de análisis sintáctico para cada gramática que reconozca el lenguaje, esta herramienta será implementada con la herramienta JavaCC, que es una herramienta de análisis léxico y sintáctico que se diferencia de otros ya que no es dependiente de una librería sino que genera el código con el algoritmo LL(1) (análisis descendente).



Analizador LR(1)

Para la construcción de analizadores sintácticos se utilizará el algoritmo de construcción de analizadores ascendentes LR(1), para ello se deberá de generar una tabla de análisis sintáctico para cada gramática que sea reconocida en IDE-Sintax.

Para ello se definen dos funciones:

1. **CERRADURA(I):**

   Función que retorna un conjunto de elementos de la forma **[B->.ɣ , b]** por cada producción de la forma **[A-> α.Bβ , a]** perteneciente al estado I donde:

   - A y B son No Terminales.
   - ɣ, α y β son conjuntos de elementos gramaticales o épsilon.
   - b se define como Primero (β). Si β contiene épsilon se le agrega Primero(a).
   - El punto . se define como el apuntador actual en la cadena.
   - La coma , se define como un separador, no perteneciente a la gramática.

2. **IR_A(I, X):**

   Función que retorna un conjunto de elementos de la forma **[A-> αX.β , a]** por cada elemento **[A-> α.Xβ , a]** perteneciente al estado I, donde:

   - X es el elemento gramatical por el cual se moverá.

3.2.1.1 Procedimiento para generación de Tabla de análisis LR(1):

1. Expandir la gramática, agregando un estado previo al estado inicial, el cual producirá al estado inicial.
2. Ejecutar la operación CERRADURA sobre el nuevo estado inicial aumentado para obtener el primer estado.
3. Ejecutar la operación IR_A para cada elemento siguiente al apuntador en el primer estado y aplicar la operación CERRADURA al resultado para obtener un estado nuevo. Si el estado ya existe no debe definirse como uno nuevo. Si el estado contiene una producción que alcanza su final debe marcarse la reducción correspondiente.
4. Repetir los pasos 2 y 3 para cada nuevo estado que se genere, deteniéndose cuando no haya más operaciones IR_A disponibles.

★ Para más información y un ejemplo completo paso a paso consultar secciones **4.7.1, 4.7.2 y 4.7.3** del libro de texto (AHO, LAM, SETHI, ULLMAN, Compiladores: principios, técnicas y herramientas. Segunda edición).

3.2.1.2 Tabla de análisis sintáctico:

La tabla de análisis sintáctico determina las acciones que debe de tomar el analizador cuando encuentra un símbolo dentro de su pila de análisis. Estas acciones construyen el comportamiento del analizador respecto al lenguaje que se quiere reconocer, el comportamiento es definido por medio de una gramática libre del contexto, que define las acciones que debe realizar el analizador, estas acciones son las siguientes:

- Desplazar: Determina la acción de desplazar al siguiente token de la pila hasta encontrar una regla de producción aplicable a la entrada.

- Reducir: Determina la acción de reducir un conjunto de tokens, que unen la parte superior de la pila según sea la regla que lo determine.

- Aceptar: El analizador ha reconocido sin errores la entrada, por lo tanto acepta la entrada y finaliza el análisis sintáctico.

- Error: El analizador ha encontrado un error en la entrada y debe realizar una acción de recuperación de errores sintácticos.

3.2.1.3 Conflictos de análisis:

Los conflictos de análisis suelen ser un problema en los analizadores SLR o LR(0) debido a que el analizador no posee la suficiente capacidad para determinar el contexto sobre la decisión que debe realizar. Los métodos LALR y Canónicos como LR(1) tendrán éxito en una colección más extensa de gramáticas debido a los caracteres LookAhead que les permiten determinar de mejor forma que acción realizar. Aun así hay gramáticas sin ambigüedad para las cuáles pueden generarse conflictos. Pero por lo general pueden ser evitadas fácilmente en las aplicaciones de los lenguajes de programación.

En caso de existir un conflicto, el libro utilizado en clase magistral en el capítulo **4.8.1,** sugiere el uso de Precedencia y asociatividad para resolver los mismos, pero por lo general pueden ser evitadas fácilmente en las aplicaciones de los lenguajes de programación cambiando la gramática.

Implementación de atributos

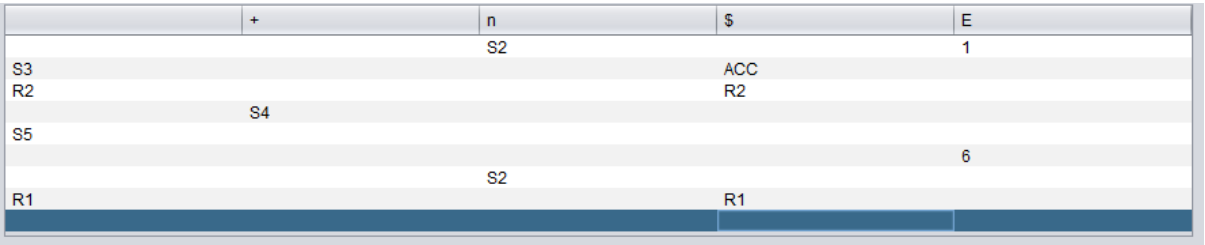
Además de realizar el análisis sintáctico luego de generar las tablas de análisis sintáctico, al momento de reconocer una cadena de entrada, se deben ejecutar las acciones asociadas con la reducción de los diferentes componentes gramaticales, el principal objetivo de estas acciones será la de implementar atributos sintetizados y heredados los cuales estarán asociados a cada componente gramatical, el resultado de ejecutar las acciones deberán de enviarse por medio de sockets al IDE (sección 3.3) en formato .txt para que puedan mostrarse gráficamente.

La implementación de los atributos utilizados dentro de las acciones del esquema de traducción se realiza a través de la pila de análisis de sintaxis (Secciones 5.4.2 y 5.4.3 del libro de texto utilizado en clase magistral)

Reportes del análisis sintáctico

Para poder verificar que el proceso de correcto del análisis sintáctico y el reconocimiento de la cadena de entrada, la solución de software debe generar los siguientes reportes:

- Registro de operaciones realizadas: Se deberá crear una lista de las operaciones realizadas durante el proceso de operación de la generación de conjuntos de elementos LR(1) y mostrarlas de manera gráfica en formato XML.
- Tablas de análisis sintáctico: luego de aplicar el método de construcción de tablas de análisis sintáctico LR(1) se deberá mostrar la representación gráfica de dichas tablas con los estados del analizador, los símbolos y las acciones a realizar con cada una de las combinaciones.



	+	n	$	E
		S2		1
S3			ACC	
R2			R2	
	S4			
S5				
				6
		S2		
R1			R1	

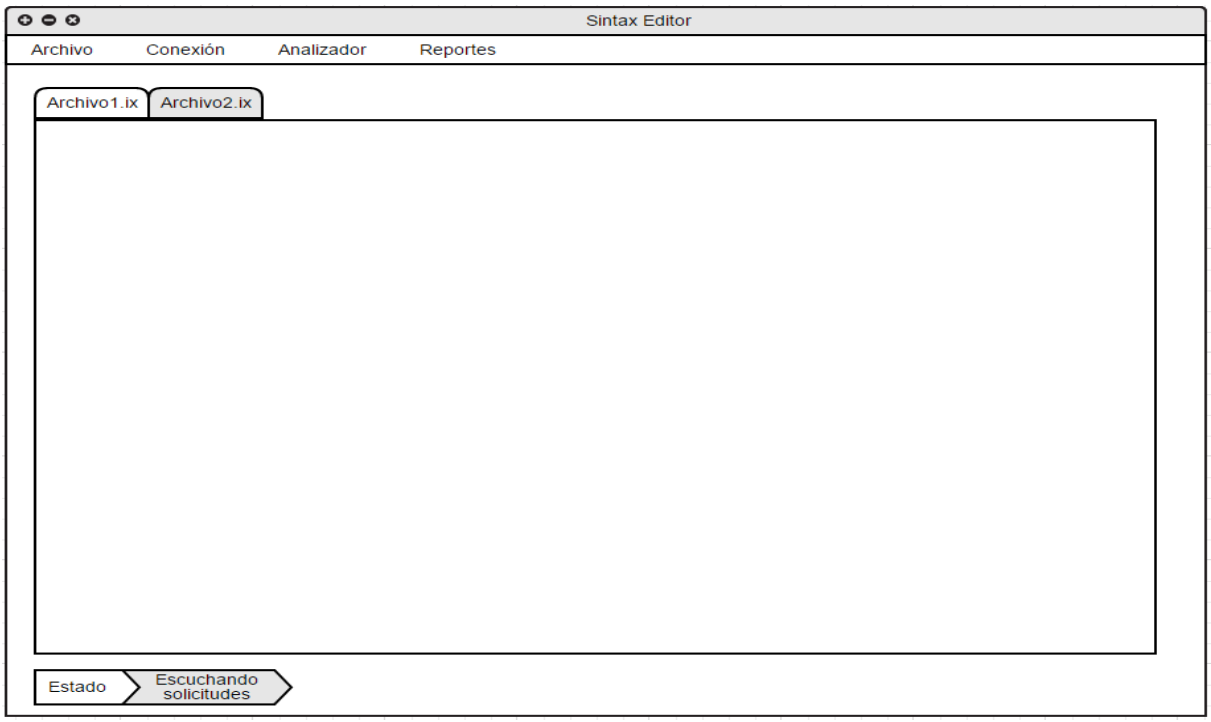
- Errores Sintácticos: Adicional al primer reporte de errores léxicos de la fase 1, la aplicación deberá generar un reporte de errores sintácticos en los que se muestre la construcción del lenguaje que no coincide con ninguna regla sintáctica, la línea y la columna en la que se encuentra el error. Este reporte de errores debe ser enviado al IDE(sección 3.3) por medio de sockets en formato XML para que pueda ser mostrado gráficamente, dentro del mismo archivo XML deberá estar el reporte de errores léxicos y sintácticos. Se debe de reconocer todos los errores sintácticos que pueden existir en la entrada, por lo cual se debe de recuperar de errores sintácticos.

Interfaz del generador de analizador de sintaxis

Las funcionalidades mínimas que deberá llevar este programa son:

- Editor de texto: el programa deberá tener un editor de texto que permita abrir los archivos de definiciones dirigidas por la sintaxis. Para que la aplicación sea más eficiente se necesita que el editor de texto pueda tener abierto más de un archivo a la vez. Los archivos que maneja el editor contendrán la gramática y sus acciones, estos estarán escritos en lenguaje IX (descrito posteriormente). Los archivos tendrán extensión .ix
- Conexión: el programa se deberá conectar con el analizador léxico por medio de apache Thrift.
- Analizador: carga a memoria el analizador sintáctico correspondiente a la gramática ingresada para analizar listados de tokens provenientes del analizador léxico su salida será el resultado de las acciones en la gramática
- Generador de reportes:
    - Registro de operaciones realizadas en Cerradura(1)
    - Tabla de análisis sintáctico LR(1).
    - Reporte de errores en la gramática proporcionada por JavaCC.

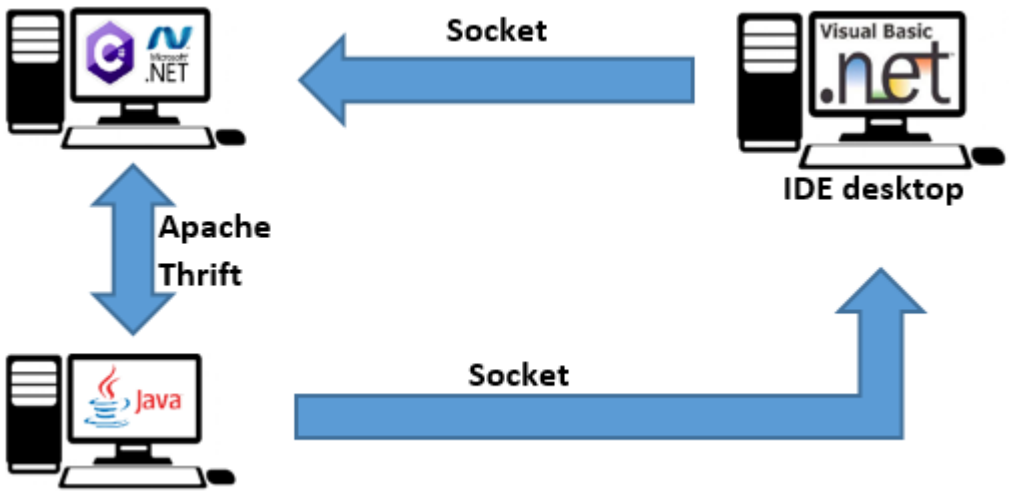
Interfaz sugerida:



3.3 IDE

El objetivo de la creación de un IDE es automatizar las herramientas ya que en esta 2da fase se integra la fase anterior y tanto el analizador léxico como el sintáctico se encuentran en plataformas diferentes. Por lo que el estudiante debe ocultar la complejidad de sus herramientas mediante un IDE, que será soportado para plataforma de escritorio

El IDE se conectan con los analizadores vía socket y mandaran las solicitudes para analizar código fuente de alto nivel así mismo obtendrán el resultado de la solicitud. Las comunicaciones serán tal y como se muestra en la imagen.



Las opciones que debe llevar el IDE de escritorio son las siguientes:

Archivo:

- Abrir archivo.
- Guardar archivo.
- Guardar archivo como.
- Archivos recientes.
- Exportar log (exporta a un archivo de texto plano el resultado de la definición dirigida por la sintaxis).

Edición:

- Buscar palabra o carácter.
- Reemplazar palabra o carácter.
- Limpiar Editor.

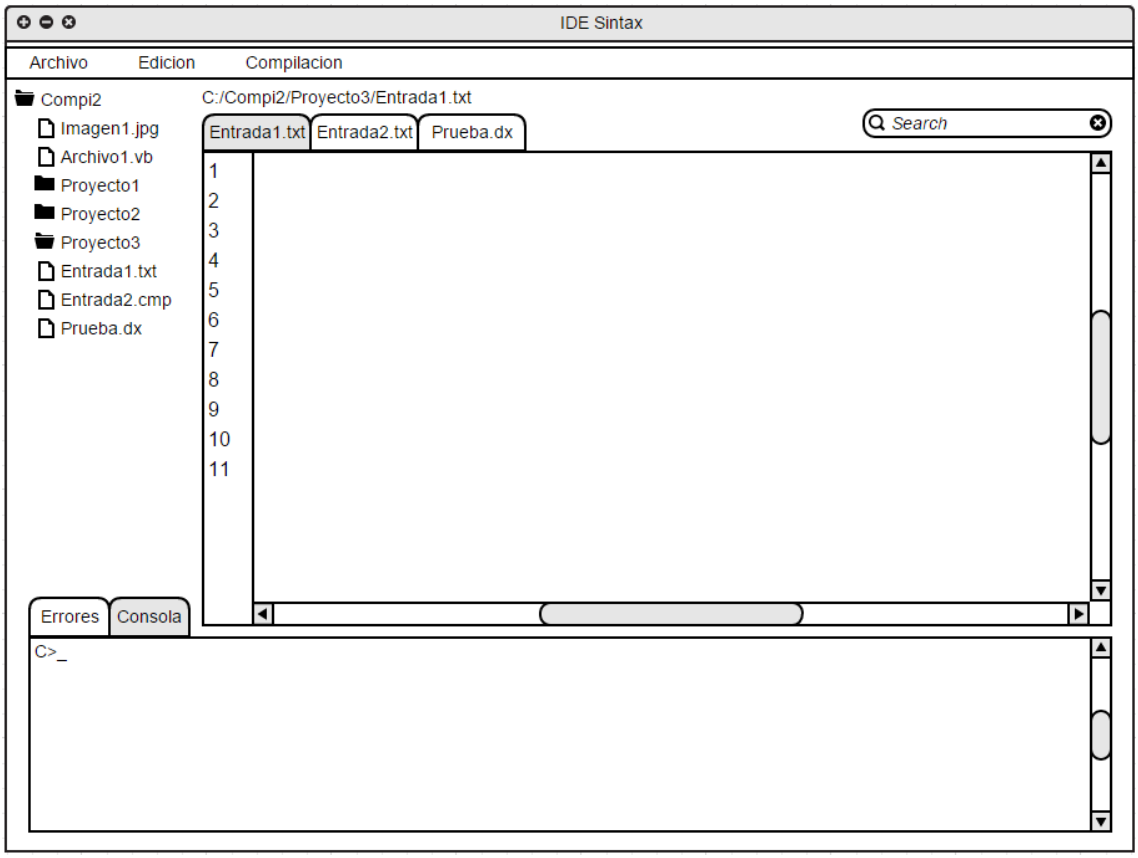
Compilación:

- Run (esta funcionalidad manda la solicitud de analizar el código a las herramientas creadas).

Las características que debe llevar el IDE son las siguientes:

- Numeración de líneas.
- Árbol de directorios.
- Multipestañas.
- Salida de errores (Errores léxicos y sintácticos de la cadena de entrada).
- Salida de resultados (Resultados de la DDS).

Interfaz sugerida



4 LENGUAJES DEL GENERADOR DE ANALIZADORES SINTÁCTICOS

La herramienta generadora de analizadores sintácticos debe utilizar un lenguaje propio de la aplicación llamado IX. El lenguaje IX define gramáticas y acciones para cada producción de la gramática, así mismo también reconoce una sección de código que estará escrita en lenguaje de alto nivel IXL, esta sección de código definirá funciones y métodos que podrán ser utilizados dentro de las acciones de la gramática. Cada uno de estos lenguajes se encuentra detallados dentro de esta sección.

4.1 DESCRIPCIÓN DEL LENGUAJE IX

IX es un lenguaje en el cual se definirán los componentes terminales y no terminales de la gramática, la definición de la gramática así como las acciones asociadas a las producciones. Este lenguaje no es sensible a mayúscula o minúsculas.

Estructura General de la gramática

La estructura general del archivo para definir la gramática contará con dos secciones, la primera sección será de las declaraciones y construcción de la gramática, la tercera será la sección de código.

```
%%
<Declaraciones y gramática>
%%
<Código>
%%
```

Declaraciones

Se realizarán cuatro tipos de declaraciones, la declaración de no terminales, terminales, precedencia y asociatividad de operadores, y del símbolo con el que inicia la gramática.

- **Terminales**

En esta sección se definirán los componentes terminales de la gramática, la definición básica de un terminal es la siguiente:

```
%%
Terminal a as int;
Terminal b as float;
Terminal c;
%%
```

Según el ejemplo anterior, el terminal *c* no tiene tipo mientras que los otros si cuentan con tipo. De igual manera se puede definir varios terminales del mismo tipo, ejemplo:

```
%%
Terminal a, b as int;
Terminal c as float;
Terminal d;
%%
```

En dado caso que no se defina un tipo para los terminales tomará el tipo ***nulo***.

- **No terminales**

En esta sección se definirán los componentes no terminales de la gramática, la definición básica de un no terminal es la siguiente:

```
%%
NoTerminal A as int;
NoTerminal B as float;
NoTerminal C;
%%
```

De igual manera se puede definir varios terminales del mismo tipo, ejemplo:

```
%%
NoTerminal A as int;
NoTerminal B,C as float;
NoTerminal D;
%%
```

En dado caso que no se defina un tipo para los no terminales tomará el tipo ***nulo***.

Tanto para los terminales y no terminales se les puede definir un tipo de dato, los tipos de datos que se manejarán serán netamente datos primitivos:

- int
- char
- float
- string
- bool

- **Precedencia y asociatividad**

En esta sección se definirán las precedencias y asociatividad de operadores que se manejarán si en dado caso aplique, de modo que al momento de realizar la tabla de análisis sintáctico no se produzca un error de desplazamiento/reducción.

  - **Precedencia**

La precedencia lo que dicta es el orden que se aplicará un operador con respecto a los otros, es decir que si una regla se produce para varios operadores se comenzará a aplicar este operador con el que tenga más precedencia, la sintaxis es la siguiente :

```
%%
RegistrarPrecedencia(num_precedencia, lista_terminales);
%%
```

Ejemplo:

```
%%
RegistrarPrecedencia(1, mas);
RegistrarPrecedencia(2, por,div);
%%
```

  - **Asociatividad**

La asociatividad se aplica cuando existen tres o más operadores, la asociatividad dicta por dónde se realizará la evaluación de dichos operadores, si de izquierda a derecha o de derecha a izquierda, para definir la asociatividad se realizará de la siguiente manera:

```
%%
RegistrarPrecedencia(num_precedencia, [Asociatividad.iz o sociatividada.der]lista_terminales);
%%
```

Ejemplo:

```
%%
RegistrarPrecedencia(1, mas);
RegistrarPrecedencia(2, Asociatividad.izq, por,div);
%%
```

- **Símbolo inicial**

En esta sección se definirá el símbolo no terminal con el que iniciará la gramática, se debe de validar que el símbolo sea un no terminal y que solo se defina una vez, ejemplo:

```
%%
Raiz = No_terminal;
%%
```

Nota: la estructura *Terminal, NoTerminal, Raiz* sólo pueden venir una vez.

Gramática

La estructura de la gramática seguirá respondiendo a la estructura siguiente, por ejemplo:

```
%%
S.Regla = E;

E.Regla = E + mas + E
        |E + por + E
        |E + div + E
        |num;

%%
```

- Para asociar una producción con sus reglas gramaticales se utiliza la sentencia ***.Regla =***.
- Para la función or se utiliza el signo de pipe |.
- Las producciones tienen como delimitador el punto y coma.
- Cada componente gramatical de una producción estará concatenado por el signo más.

Estructura general:

```
%%
```

```
E.Regla = N1 + N2 + N3 + ... + Nk
| M1 + M2 + M3 + … + Mk
.
.
.
|X1 + X2 + X3 + … + Xk;
%%
```

Donde *N, M y X* pueden ser terminales o no terminales.

Ejemplo 1:

```
%%
Terminal palabra as string;
NoTerminal L;
Raiz = L;
L.Regla = L + palabra
|palabra;
%%
```

Ejemplo 2:

En el siguiente ejemplo se puede observar ya toda la estructura unificada (sin acciones).

```
%%
NoTerminal num as int;
NoTerminal mas, menos, por, div;
Terminal S, E;
RegistrarPrecedencia(2, Asociatividad.izq, por, div);
RegistrarPrecedencia(1, mas);
Raiz = S;

S.Regla = E;
E.Regla = E + mas + E
```

```
|E + por + E
|E + div E
|menos + num
|num;
%%
```

Para este ejemplo (operaciones aritméticas), en azul se han marcado las palabras reservadas y en rojo los tipos de datos. La gramática es ambigua, pero para eso se declara la precedencia, para este caso reducirá primero por los terminales *por* y *div*, y con esto se resuelve conflictos en la tabla de analizador sintáctico.

Nota: el anterior ejemplo es un ejemplo de una gramática sin acciones, al final del apartado 4 encontrará ejemplos completos de gramáticas con sus acciones.

Acciones de la gramática

Para generar una definición dirigida por la sintaxis o traducción dirigida por la sintaxis es necesario que tengamos una gramática y que insertemos acciones entre los componentes gramaticales. Para declarar una acción entre los componentes gramaticales utilizaremos la siguiente sintaxis:

```
%%
A.Regla = A <:: acción/acciones ::> + B <:: acción/acciones ::>
|B <:: acción/acciones ::>;
%%
```

Nótese que las acciones pueden venir intercaladas entre los componentes gramaticales sin distinción dentro de las llaves **<:: ::>** pueden venir más de una sentencia. Dentro de cada acción se puede tener realizar asignaciones a atributos, llamadas a métodos o funciones propias de la sección de código y operaciones aritméticas y lógicas. Pueden venir varias acciones o sentencias y su delimitador será el **punto y coma (;)**, por ejemplo:

```
%%
A.Regla = A <:: acción1; accion2;::> + B <:: acción3; accion4: ::>
|B<:: acción5; accion6; ::> ;
%%
```

4.1.1.1 Atributos sintetizados

El generador de gramáticas del IDE podrá implementar el uso de atributos sintetizados y dichos atributos se manejan de la siguiente manera:

- Para acceder a un atributo se hará mediante la siguiente sentencia:
    - RESULT#
        - Donde # indica el número asociada al componente gramatical de la producción.
- Para acceder a un atributo se hará mediante la siguiente forma:
    - RESULT#.atributo
        - Donde # indica el número asociada al componente gramatical de la producción.
        - Donde *atributo* es el nombre del atributo.
- RESULT0 hace referencia al no terminal al cual se asocian las producciones.
- Cuando se hace referencia a un terminal simplemente se colocará RESULT# y si cuenta con un valor retornado del archivo de tokens de la primera práctica entonces ese valor hará referencia.
    - Donde # indica el número asociada al componente gramatical de la producción.

  Ejemplo:

```
%%
NoTerminal A;
Terminal num as int;

A.Regla = A + num<:: RESULT0.valor = RESULT1.valor; RESULT0.valor = RESULT0.valor + RESULT2; ::>
| num <:: RESULT0.valor = RESULT1;::>;
%%
```

Cabe destacar que se puede realizar operaciones aritméticas elementales así como llamada a métodos y funciones que están declaradas en la sección de código, por ejemplo:

```
%%
NoTerminal A;
Terminal num as int;
Raiz = A;

A.Regla = A + num<:: RESULT0.valor = RESULT1.valor / suma(RESULT2); ::>
|num<:: RESULT0.valor = RESULT1;::>;
```

```
%%


%%
funcion suma(val as int) as int
c as int
c = val +1
return c
end function
%%
```

El ejemplo anterior es una simple llamada a una función, este ejemplo contiene ya una pequeña función de la sección de código que será explicado más adelante.

4.1.1.2 Atributos heredados

El generador de gramáticas del IDE podrá implementar el uso de atributos heredados y dichos atributos se manejaran de la siguiente manera:

- Para acceder a un atributo se hará mediante la siguiente sentencia:
  - RESULT#
    - Donde # indica el número asociada al componente gramatical de la producción.
- Para acceder a un atributo se hará mediante la siguiente forma:
  - RESULT#.atributo**H**
    - Donde # indica el número asociada al componente gramatical de la producción.
    - Donde *atributo* es el nombre del atributo.
    - El nombre de cada atributo debe de llevar el sufijo **H**.
- RESULT0 hace referencia al no terminal al cual se asocian las producciones.
- Cuando se hace referencia a un terminal simplemente se colocará RESULT# y si cuenta con un valor retornado del archivo de tokens de la primera práctica entonces ese valor hará referencia.
  - Donde # indica el número asociada al componente gramatical de la producción.

Ejemplo:

```
%%
NoTerminal D, T;
Terminal coma;
```

```
Terminal tipo as string;
Raiz = D;

D.Regla = T + L <::RESULT2.tipoH = RESULT1.tipo;::>;

T.Regla = int <::RESULT0.tipo = "integer";::>
|float <::RESULT0.tipo = "float";::>;

L.Regla = L + coma + id <::RESULT1.tipoH = RESULT0.valor;
agregarTipo(RESULT3, RESULT0.tipoH);::>;

L.Regla = id <::agregarTipo(RESULT1, RESULT0.tipoH);::>;

%%
sub agregarTipo(var as string, tipo as string)
write(var + ": " +tipo);
end sub
%%
```

El ejemplo anterior es de la declaración de variables con su tipo, mediante atributos heredados, este ejemplo contiene ya un pequeño método de la sección de código que será explicado más adelante.

Sección de código

La sección de código es la última sección que estará en el archivo .ix, en esta sección se implementarán funciones y métodos que serán utilizados por las acciones que se encuentran en la gramática, dicho código será estructurado. Esta sección estará escrita en un lenguaje de alto nivel IXL.

Ejemplo de la sección de código de la gramática

```
%%
sub metodo1(x as int, y as int)
```

```
a as int
a = 0
if x ==y then
a = x * y
end if
end sub
%%
```

4.2 DESCRIPCIÓN DEL LENGUAJE IXL

IXL es un lenguaje de alto nivel estructurado el cual es capaz de declarar variables y definir métodos y funciones. Dentro de esta sección de disposiciones generales se considera que se refiere a cualquiera de los lenguajes de alto nivel, sin distinción alguna entre ellos.

Anotaciones iniciales

- **Sensibilidad a mayúsculas**

El lenguaje de alto nivel **no** es case sensitive, es decir que no existe diferencia entre escribir un identificador, palabra reservada o **cualquier elemento del lenguaje** con mayúsculas, minúsculas o ambas a la vez.

- **Identificadores**

Un identificador es el nombre propio que recibe un elemento del lenguaje de alto nivel, es decir una variable, una clase, un procedimiento, etc. Como norma general un identificador siempre comienza con una letra seguida opcionalmente de más letras, números o guiones bajos '_' .

Dos o más elementos del mismo tipo no pueden tener el mismo identificador, pero elementos de diferente tipo si puede tener el mismo identificador. Esto quiere decir que podemos definir una variable con el mismo nombre que una clase o que un procedimiento.

- **Comentarios**

Dentro del lenguaje de alto nivel se pueden definir comentarios para entender mejor el código del programa, estos comentarios no afectan el flujo de ejecución. Los comentarios pueden venir en cualquier parte de la cadena de entrada.

Podemos definir dos tipos de comentarios:

- **Comentarios de una línea:**

```
//Este es un comentario de una línea
```

- **Comentarios de varias líneas:**

```
/* Este es un ejemplo
```

```
de un comentario de
varias líneas */
```

Tipos de datos

Entendemos a un tipo de dato como la restricción que se le asigna a un miembro de una clase sobre los datos que puede contener y las acciones que se pueden realizar sobre él. Ambos lenguajes de alto nivel soportan los tipos de datos primitivos. La tipificación de los lenguajes de alto nivel es tipificación fuerte, se deben declarar los tipos de datos asociados a los elementos del lenguaje y este tipo de datos no cambiará en todo el código.

Tipos de datos aceptados

- **int:** tipo de dato que acepta valores numéricos enteros en base decimal (base 10).
  - Ejemplos de int: -85, 0, 15, 1589.
- **double:** tipo de dato que acepta valores numéricos con punto flotante de doble precisión en base decimal (base 10).
  - Ejemplos de double: -5.68, 3.141592, 0.008.
- **string:** tipo de dato que acepta cadenas de caracteres alfanuméricas. Un dato de tipo string debe estar escrito entre comillas dobles.
  - Ejemplo de string: “Este es un ejemplo de una cadena string”.
- **char:** tipo de dato que acepta un carácter alfanumérico. Un dato de tipo char puede estar escrito entre comillas simples o puede recibir un número entero entre 0 y 255, en este caso el número será convertido al carácter equivalente del código ASCII. Un tipo de dato char también puede ser utilizado como un dato numérico, en este caso el carácter deberá ser convertido a su código ASCII correspondiente, la decisión de la forma de usar un tipo de dato char se tomará según el ámbito en el que se encuentre.
  - Ejemplo de char: ‘a’, ‘1’, 23, ‘?’, 55.
- **bool:** tipo de dato que acepta un valor lógico verdadero o falso.
  - Datos aceptados como valores lógicos verdaderos: true, 1.
  - Datos aceptados como valores lógicos falsos: false, 0.

Operaciones aritméticas

Una operación aritmética es un conjunto de reglas que permiten obtener otras cantidades o expresiones a partir de datos específicos. Cuando el resultado de una operación aritmética sobrepase el rango establecido para los tipos de datos deberá mostrarse un error en tiempo de ejecución. A continuación se definen las operaciones aritméticas soportadas por el lenguaje.

4.2.1.1 Suma:

Operación aritmética que consiste en reunir varias cantidades (sumandos) en una sola (la suma). El operador de la suma es el signo más **+**

Especificaciones sobre la suma:

- Al sumar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- Al sumar dos datos de tipo carácter (char, string) el resultado será la concatenación de ambos datos.

- Al sumar un dato numérico con un dato de tipo carácter el resultado será la concatenación de del dato de tipo carácter y la conversión a cadena del dato numérico.
- Al sumar dos datos de tipo lógico (bool) el resultado será un dato lógico, en este caso utilizaremos la suma como la operación lógica or.

Operandos	Tipo de dato resultante	Ejemplos
int + double double + int double + char char + double bool + double double + bool double + double	double	5+4.5 = 9.5 7.8+3 = 10.8 15.3 + 'a' = 112.3 'b' + 2.7 = 100.7 true + 1.2 = 2.2 4.5 + false = 4.5 3.56+2.3 = 5.86
int + char char + int bool + int int + bool int + int	int	7 + 'c' = 106 'C' + 7 = 74 4 + true = 5 4 + false = 4 4 + 5 = 9
string + int string + double double + string int + string	string	"hola" + 2 = "hola2" "hola"+3.5 = "hola3.5" 4.5 + "hola" = "4.5hola" 8+"hola" = "8hola"
string + char char + string string + string	string	"hola"+'t' = "holat" 'u'+"hola" = "uhola" "hola"+ "mundo" = "holamundo"
bool + bool	bool	1 + false = true = 1 true + true = true = 1 0 + 1 = 1 = true false + 0 = false = 0

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.2.1.2 Resta:

Operación aritmética que consiste en quitar una cantidad (sustraendo) de otra (minuendo) para averiguar la diferencia entre las dos. El operador de la resta es el signo menos -

Especificaciones sobre la resta:

- Al restar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible restar datos numéricos con tipos de datos de tipo caracter (string)
- No es posible restar tipos de datos caracter (char, string) entre sí.
- No es posible restar tipos de datos lógicos (bool) entre sí.

Operandos	Tipo de dato resultante	Ejemplos
int - double double - int double - char char - double bool - double double - bool double - double	double	5-4.5 = 0.5 7.8-3 = 4.8 15.3 - ‘a’ = -81.7 ‘b’ - 2.7 = 95.3 true - 1.2 = -0.2 4.5 - false = 4.5 3.56 - 2.3 = 1.26
int - char char - int bool - int int - bool int - int	int	7 - ‘c’ = -92 ‘C’ - 7 = 60 4 - true = 3 4 - false = 4 4 - 5 = -1

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.2.1.3 Multiplicación:

Operación aritmética que operación aritmética que consiste en sumar un número (multiplicando) tantas veces como indica otro número (multiplicador). El operador de la multiplicación es el asterisco *****

Especificaciones sobre la multiplicación:

- Al multiplicar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible multiplicar datos numéricos con tipos de datos caracter (string)
- No es posible multiplicar tipos de datos caracter (char, string) entre sí.
- Al multiplicar dos datos de tipo lógico (bool) el resultado será un dato lógico, en este caso usaremos la multiplicación como la operación AND entre ambos datos.

Operandos	Tipo de dato resultante	Ejemplos
int * double double * int double * char char * double bool * double double * bool double * double	double	5*4.5 = 22.5 7.8*3 = 23.4 15.3 * ‘a’ = 1484.1 ‘b’ * 2.7 = 264.6 true * 1.2 = 1.2 4.5 * false = 0 3.56 * 2.3 = 8.188
int * char char * int bool * int int * bool int * int	int	7 * ‘c’ = 693 ‘C’ * 7 = 469 4 * true = 4 4 * false = 0 4 * 5 = 20

bool * bool	bool	true * 0 = 0 = false true * 1 = 1 = true 0 * false = 0 = false 1 * false = 0 = false

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.2.1.4 División:

Operación aritmética que consiste en partir un todo en varias partes, al todo se le conoce como dividendo, al total de partes se le llama divisor y el resultado recibe el nombre de cociente. El operador de la división es la diagonal **/**

Especificaciones sobre la división:

- Al dividir dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible dividir datos numéricos con tipos de datos de tipo caracter (string)
- No es posible dividir tipos de datos caracter (char, string) entre sí.
- No es posible dividir tipos de datos lógicos (bool) entre sí.
- Al dividir un dato numérico entre 0 deberá arrojarse un error de ejecución.

Operandos	**Tipo de dato resultante**	**Ejemplos**
int / double double / int double / char char / double bool / double double / bool double / double int / char char / int bool / int int / bool int / int	double	5/4.5 = 1.11111 7.8/3 = 2.6 15.3 / ‘a’ =0.1577 ‘b’ / 2.7 = 28.8889 true / 1.2 = 0.8333 4.5 / false = error 3.56 / 2.3 = 1.5478 7 / ‘c’ = 0.7070 ‘C’ / 7 = 9.5714 4 / true = 4.0 4 / false = error 4 / 5 = 0.8

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.2.1.5 Potencia:

Operación aritmética que consiste en multiplicar varias veces un mismo factor. El operador de la potencia es el acento circunflejo **^**

Especificaciones sobre la potencia:

- Al potenciar dos datos numéricos (int, double, char, bool) el resultado será numérico.
- No es posible potenciar datos numéricos con tipos de datos de tipo caracter (string)
- No es posible potenciar tipos de datos caracter (char, string) entre sí.
- No es posible potenciar tipos de datos lógicos (bool) entre sí.

Operandos	**Tipo de dato resultante**	**Ejemplos**

int ^ double double ^ int double ^ char char ^ double bool ^ double double ^ bool double ^ double	double	5^4.5 = 1397.54 7.8^3 = 474.55 15.3 ^ 'a' = <<fuera de rango>> 'b' ^ 2.7 = 237853.96 true ^ 1.2 = 1.0 4.5 ^ false = 1.0 3.56 ^ -2.3 = 0.0539
int ^ char char ^ int bool ^ int int ^ bool int ^ int	int	7 ^ 'c' = <<fuera de rango>> 'C' ^ 7 = 6060711605323 4 ^ true = 4 4 ^ false = 1 4 ^ 5 = 1024

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.2.1.6 Aumento:

Operación aritmética que consiste en añadir una unidad a un dato numérico. El aumento es una operación de un solo operando. El aumento sólo podrá venir del lado derecho de un dato. El operador del aumento es el doble signo más **++**

Especificaciones sobre el aumento:

- Al aumentar un tipo de dato numérico (int, double, char) el resultado será numérico.
- No es posible aumentar tipos de datos caracter (string).
- No es posible aumentar tipos de datos lógicos (bool).
- El aumento podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros, atributos, etc).

Operandos	Tipo de dato resultante	Ejemplos
double++	double	4.56++ = 4.57
int ++ char ++	int	15++ = 16 'a'++ = 98 = 'b'

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

4.2.1.7 Decremento:

Operación aritmética que consiste en quitar una unidad a un dato numérico. El decremento es una operación de un solo operando. El decremento sólo podrá venir del lado derecho de un dato. El operador del decremento es el doble signo menos **--**

Especificaciones sobre el decremento:

- Al decrementar un tipo de dato numérico (int, double, char) el resultado será numérico.
- No es posible decrementar tipos de datos caracter (string).
- No es posible decrementar tipos de datos lógicos (bool).

- El decremento podrá realizarse sobre números o sobre identificadores de tipo numérico (variables, parámetros, atributos, etc).

Operandos	Tipo de dato resultante	Ejemplos
double--	double	4.56-- = 3.57
int -- char --	int	15-- = 14 'b'-- = 97 = 'a'

Cualquier otra combinación es inválida y deberá arrojar un error de semántica.

Precedencia de operadores

Para saber el orden jerárquico de las operaciones aritméticas se define la siguiente precedencia de operadores. Todas las operaciones aritméticas tienen asociatividad izquierda.

- **Precedencia de operadores aritméticos**
  La precedencia de los operadores va de menor a mayor según su aparición en la siguiente tabla.

Operador
+ -
*** /**
^
++ --

- **Signos de agrupación**
  Para dar orden y jerarquía a ciertas operaciones aritméticas se utilizan los signos de agrupación. A continuación se definen los signos de agrupación válidos. La precedencia de los signos de agrupación va de menor a mayor según su aparición en la tabla.

Signos de agrupación	Ejemplo de uso
{ }	{4^[6*(8-7)]+5}-2
[ ]	4^[6*(8-7)]+5
( )	6*(8-7)

4.2.1.8 Operaciones relacionales

Una operación relacional es una operación de comparación entre dos valores, el cual siempre devuelve un valor de tipo lógico (bool) según el resultado de la comparación. Una operación relacional es binaria, es decir tiene dos operandos siempre.

Especificaciones sobre las operaciones relacionales:

- Es válido comparar datos numéricos (int, double) entre sí, la comparación se realizará utilizando el valor numérico con signo de cada dato.
- Es válido comparar cadenas de caracteres (char, string) entre sí, la comparación se realizará sobre el resultado de sumar el código ASCII de cada uno de los caracteres que forman la cadena.
- Es válido comparar datos numéricos con datos de carácter (‘char’) en este caso se hará la comparación del valor numérico con signo del primero con el valor del código ASCII del segundo.
- No es válido comparar cadenas de caracteres (string) con datos numéricos.
- No es válido comparar cadenas de caracteres (string) o datos numéricos (double, int, char) con valores lógicos (bool).
- No es válido comparar valores lógicos (bool) entre sí.

Operador relacional	Ejemplos de uso
>	5>4 = true 4.5>6 = false “abc” > “abc” = false ‘a’ > “a” = false 97 > ‘a’ = false
<	5<4 = false 4.5<6 = true “abc” < “abc” = false ‘a’ < “a” = false 97 < ‘a’ = false
>=	5>=4 = true 4.5>=6 = false “abc” >= “abc” = true ‘a’ >= “a” = true 97 >= ‘a’ = true
<=	5<=4 = false 4.5<=6 = true “abc” <= “abc” = true ‘a’ <= “a” = true 97 <= ‘a’ = true
==	5==4 = false 4.5==6 = false “abc” == “abc” = true ‘a’ == “a” = true

	97 == ‘a’ = true
!=	5!=4 = true 4.5!=6 = true “abc” != “abc” = false ‘a’ != “a” = false 97 != ‘a’ = false

4.2.1.9 Operaciones lógicos

Las operaciones lógicas son expresiones matemáticas cuyo resultado es un valor lógico (bool). Las operaciones lógicas se basan en el álgebra de Boole.

Especificaciones sobre las operaciones lógicas:

- No es válido realizar una operación lógica sobre dos datos de tipo numérico (int, double, char)
- No es válido realizar una operación lógica sobre dos datos de tipo carácter (string)
- No es válido realizar una operación lógica sobre un dato de tipo numérico con un dato de tipo carácter.
- No es válido realizar una operación lógica sobre un dato de tipo numérico o carácter sobre un tipo de dato lógico.
- Si es válido realizar una operación lógica sobre dos datos de tipo lógico.
- La operación lógica NOT tiene sólo un operando, el resto de operaciones lógicas debe tener dos operandos.
- La operación lógica NOT tiene asociatividad derecha, el resto de operaciones lógicas tienen asociatividad izquierda.

<table>
<tr><th>Operación lógica</th><th>Operador</th><th>Tabla de verdad</th></tr>
<tr><td>OR</td><td>Or</td><td>
<table>
<tr><th>Op1</th><th>Op2</th><th>Op1 Or Op2</th></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>XOR</td><td>Xor</td><td>
<table>
<tr><th>Op1</th><th>Op2</th><th>Op1 Xor Op2</th></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td></td><td><table>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table></td></tr>
<tr><td>AND</td><td>And</td><td><table>
<tr><th>Op1</th><th>Op2</th><th>Op1 And Op2</th></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
</table></td></tr>
<tr><td>NOT</td><td>Not</td><td><table>
<tr><th>Op1</th><th>Not Op1</th></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td></tr>
</table></td></tr>
</table>

Precedencia de operadores lógicos y signos de agrupación

Para saber el orden jerárquico que tienen las operaciones lógicas definimos la precedencia de operadores, consideramos la precedencia de menor a mayor según el orden de aparición del operador lógico en la tabla de precedencia. El único símbolo de agrupación válido para operaciones lógicas son los paréntesis **( )** los cuales sirven para dar jerarquía y orden a las operaciones lógicas.

Operación lógica
OR
XOR
AND
NOT

Sintaxis del lenguaje IXL

La sintaxis del lenguaje IXL describe las combinaciones posibles de los símbolos que forman el programa sintácticamente correcto. A continuación se define la sintaxis del lenguaje:

4.2.1.10 Declaración de variables

Existen dos tipos de variables en la sección de código, variables globales y locales, cada variable debe de tener su tipo asociado.

Estructura de declaración:

```
variable as tipo
lista_de_variables as tipo
```

Ejemplo:

```
variable as int
decimal1, decimal2 as float
```

4.2.1.11 Declaración de vectores y matrices

Existen dos tipos de arreglos en la sección de código, arreglos de una dimensión (vectores) y arreglos de varias dimensiones (matrices), cada arreglo debe de tener su tipo asociado.

Estructura de declaración:

```
vector (numelemnto) as tipo
matriz (numelentox)(numelementoy) as tipo
```

Ejemplo:

```
vector (2) as int
matriz (4)(2) as float
```

Nota: en la definición de los números de elementos pueden venir valores puntuales, una operación aritmética, variables o incluso llamadas a funciones que retornan un valor entero.

4.2.1.12 Asignación

Para asignar un valor a una variable dentro del lenguaje debemos hacerlo por medio del signo = , las asignaciones de valores únicamente pueden hacerse en la

declaración de variables, dentro de las producciones o dentro de los métodos o funciones.

La asignación debe seguir la siguiente sintaxis:

```
variable = expresión

vector(posición) = expresión

matriz(posición)(posición) = expresión
```

Ejemplo:

```
a = 100

vec (1) = 10

mat (0)(0) = 1
```

4.2.1.13 Métodos y funciones

Un método es un conjunto de instrucciones que se ejecutan sin retornar un valor, una función es un método que tiene un valor de retorno.

- **Métodos**

Los métodos responderán a la siguiente estructura:

```
sub nombreMetodo(parametros)

//código

end sub
```

Ejemplo:

```
sub metodo(a as int)

b as float

b = a * 2

end sub
```

- **Funciones y sentencia return**

Las funciones responderán bajo la siguiente estructura y como mínimo deben de contar con una sentencia de retorno.

```
function nombrefuncion(parametros) as tipo
//código
end function
```

Ejemplo:

```
function suma(a as float, b as float) as float
c as float
c = a + b
return c
end function
```

4.2.1.14 Sentencias de flujo

- **IF**
  Sentencia que tiene la finalidad de crear un flujo en el programa, dependiendo de una condición que es establecida.

  La sintaxis de la sentencia if es:

```
if condicion then
//sentencias
end if
```

Ejemplo:

```
if x >6 then
x = x * 2
end if
```

- **IF ELSE**
  Sentencia que contiene un set de instrucciones alternativo al flujo que el programa debe seguir basado en una condición establecida.

  La sintaxis de la sentencia if else es:

```
if condicion then
//sentencias
else
//sentencias
end if
```

Ejemplo:

```
if x == y then
z = x * y
else
z = 0
end if
```

- **IF ELSEIF**
  Sentencia que engloba acciones diferentes, para condiciones distintas.

  La sintaxis de la sentencia if elseif es:

```
if condicion1 then
//sentencias
elseif condicion2 then
//sentencias
.
.
.
elseif condicionN then
//sentencias
end if
```

Ejemplo:

```
if x==y then
```

```
z = x*y
elseif y==z then
x = 0
elseif z == 0 then
y =0
end if
```

- **SWITCH**
  Es una sentencia que agrupa flujos alternativos sobre un solo caso, en el caso que una condición se cumpla, dentro de este debe ejecutarse todas las instrucciones contenidas hasta que encuentre la sentencia de salida.

  - **Sentencia de salida**

  Esta sentencia indicará cuándo debe terminar la ejecución del flujo dentro de las condiciones establecidas por el programador, la sentencia tendrá la siguiente forma:

```
break
```

Sintaxis

La sintaxis de la sentencia switch es:

```
switch expresión
case valor1
//sentencia1
case valor2
//sentencia2
.
.
.
case valorN
//sentenciaN
case default
//sentenciaDefault
```

```
end switch
```

Ejemplo:

```
switch x
case 1
x = 2
break
case 2
y = x + 2
case 3
z = 0
break
case default
y= 0
end switch
```

Nota: la sentencia ***break*** es considerada una sentencia más y puede o no venir, si en dado caso no viene sigue ejecutando.

4.2.1.15 Bucles

- **WHILE**
  Ciclo que se ejecuta mientras la condición o expresión sea verdadera.

  La estructura es la siguiente:

```
while condición
//sentencias
end while
```

Ejemplo:

```
while var1 > var2
var1 = var2*2
end while
```

- **DO WHILE**
  Ciclo que se ejecuta al menos una vez, luego la se ejecuta mientras la condición o expresión sea verdadera.

  La estructura es la siguiente:

```
do
//sentencias
while condición
```

  Ejemplo:

```
do
x ++
while x < 10
```

- **LOOP**
  Ciclo que se ejecuta en un número infinito de iteraciones, hasta que exista una sentencia de escape.

  La estructura es la siguiente:

```
loop
//sentencias
end loop
```

  Ejemplo:

```
loop
if x == 0 then
```

```
break

else

x --

end if

end loop
```

- **FOR**
  Ciclo que se ejecuta el número de veces, aumentando de valor según sea lo que se defina en el aumento, desde un valor inicial hasta un valor final, la variable de control debe de estar declarada con anterioridad y debe de ser un entero.

  La estructura es la siguiente:

```
for variable = valorinicial to valorfinal [step valoraumennto]

//setentencias

next
```

La sentencia *step* puede o no venir y dicta el aumento que tendrá en cada iteración del ciclo for.

Ejemplo 1:

```
for x = 0 to 10 step 2

y = y *x

next
```

En el anterior ejemplo es un ciclo que irá de dos en dos su aumento.

Ejemplo 2:

```
for x = 10 to 0 step -1

y = y *x

next
```

En el anterior ejemplo es un ciclo for invertido.

Ejemplo 3:

```
for x = 0 to 10
y = y *x
next
```

En el anterior ejemplo es un ciclo for que irá de uno en uno su aumento dado que no tiene la sentencia ***step***.

4.2.1.16 Funciones primitivas del lenguaje

- **Write**

Método que muestra en consola una expresión de cualquier tipo

La estructura es la siguiente:

```
write(expresión)
```

Ejemplo:

```
x as int
x = 10
write(x)
write("hola mundo")
```

Ejemplo

```
%%
Terminal num as float;
Terminal mas, menos, por, div;

NoTerminal E as float;
NoTerminal S;

RegistrarPrecedencia(2, Asociatividad.izq, por, div);
RegistrarPrecedencia(1, mas);

Raiz = S;

S.Regla = E;

E.Regla = E + mas + E <::RESULT0.valor = suma(RESULT1.valor, RESULT2.valor);::>
|E + por + E  <::RESULT0.valor = multiplicacion (RESULT1.valor, RESULT2.valor);::>
|E + div E <::RESULT0.valor = division (RESULT1.valor, RESULT2.valor);::>
|menos + num <::RESULT0.valor = -RESULT2;::>
|num <::RESULT0.valor = RESULT1;::>;
%%
function suma (a as float, b as float) as float
        c as float
        c = a + b
        write("resultado " + c)
        return c
end function

function multiplicacion (a as float, b as float) as float
        c as float
        c = a * b
        write("resultado " + c)
        return c
end function

function division (a as float, b as float) as float
        c as float
        c = a / b
        write("resultado " + c)
        return c
end function
%%
```

5 ENTREGABLES Y RESTRICCIONES

5.1 RESTRICCIONES

- La aplicación será desarrollada sobre el lenguaje de programación Java para el generador de gramáticas y VB.Net para el IDE.
- Es obligatoria la integración de la práctica con el proyecto.
- El proyecto es de manera individual.
- Para los archivos .ix se utilizará JavaCC (análisis descendente).
- Copias de código fuente o de gramáticas serán merecedoras de una nota de 0 puntos y los responsables serán reportados al catedrático de la sección, así como a la Escuela de Ciencias y Sistemas.

5.2 MÉTODO DE CALIFICACIÓN

Para la calificación del proyecto se ingresará un archivo .dx en la aplicación de la práctica y un archivo .ix a la nueva aplicación. De esta forma se generarán los analizadores correspondientes.

En esta sección se calificarán los reportes generados durante la generación del analizador sintáctico, entre ellos:

- Registro de operaciones realizadas
- Tabla de análisis Sintáctico
- Reporte de errores sintácticos

Luego se ingresará un archivo desde el IDE, el cual debe recorrer todo el proceso de análisis, incluyendo el análisis léxico y sintáctico. Al finalizar se comprobará que se hayan ejecutado las acciones de la DDS mediante la respuesta que regresará al IDE y el archivo que debe generar el mismo.

5.3 MANEJO DE ERRORES

La aplicación debe de ser capaz de manejar errores y recuperarse de los mismos en cada fase de análisis de la compilación, para cada error se debe de indicar la línea, columna y descripción del mismo y tipo de error, los tipos de errores son los siguientes:

- Errores léxicos.
- Errores sintácticos.
- Errores semánticos.

5.4 REQUERIMIENTOS MÍNIMOS

Los requerimientos mínimos son funcionalidades que garantizan una ejecución básica del sistema, para tener derecho de calificación se debe de cumplir con los siguientes requerimientos:

- Integración con la práctica (comunicación entre los analizadores).
- Tabla de análisis sintáctico LR(1).
- Atributos sintetizados y estructura total de la gramática
- IF ELSEIF ELSE
- WHILE
- SWITCH
- FOR
- Métodos
- Funciones
- Método primitivo write(*expresión*)

5.5 ENTREGABLES

- Aplicación funcional.
- Código fuente.
- Gramática escrita en JavaCC.
- Manual técnico.
- Manual de usuario.

Fecha de entrega: lunes 28 de marzo de 2016