Instituto Tecnológico y de Estudios Superiores de Monterrey

TC3002B Desarrollo de Aplicaciones avanzadas de Ciencias Computacionales Gpo 502

Patito – Entrega #2

José Eduardo de Valle Lara A01734957

Profesora:

Elda Guadalupe Quiroga González

Contenido

Etapa 0

Tokens y Expresiones Regulares

Se agruparon los tokens con sus respectivas expresiones regulares en una misma sección para simplificar el documento. Las expresiones regulares de las palabras reservadas y algunos operadores son el mismo token ya que son patrones que no cambian, son fijos.

Token	**Expresión Regular**	**Explicación**
ID	[a-zA-Z][a-zA-Z0-9]*	Se usa para nombrar variables y funciones, debe comenzar con una letra y puede ser seguido por cualquier combinación de letras y números
Constantes: valores fijos que no cambian durante la ejecución del programa		
Enteros (cte_int)	[0-9]+	Debe haber al menos un número
Flotantes (cte_float)	[0-9]+\.[0-9]+	Empiezan con al menos uno o muchos números seguido de un punto “.” y de al menos un número después del punto
Strings (cte.string)	\"[^"]*\"	Empiezan con una comilla doble “ seguida de cero o más repeticiones de cualquier caracter que no sea una comilla doble y finaliza con una comilla doble
Operadores y símbolos: se usan para realizar operaciones, comparaciones y agrupaciones		
+	\+	Se usó un backslash \ para indicar la expresión regular de + ya que en expresiones regulares se refiere a uno o más elementos
-	-	
*	*	Se usó un backslash \ para indicar la expresión regular de * ya que en expresiones regulares se refiere a cero o más elementos
/	/	
>	>	

<	<	
!=	!=	
=	=	
;	;	
:	:	
,	,	
(	\(	Se usó un backslash \ para indicar la expresión regular de (porque se usa para agrupaciones en expresiones regulares
)	\)	Se usó un backslash \ para indicar la expresión regular de) porque se usa para agrupaciones en expresiones regulares
{	\{	Se usó un backslash \ para indicar la expresión regular de { porque se usa para cuantificadores en expresiones regulares
}	\}	Se usó un backslash \ para indicar la expresión regular de } porque se usa para cuantificadores en expresiones regulares
Palabras reservadas: identificadores con un significado especial que no pueden usarse como nombres de variables o funciones		
program	program	
main	main	
end	end	
var	var	
int	int	
float	float	
void	void	
if	if	
else	else	
while	while	

do	do	
print	print	

Definición de la gramática libre de contexto

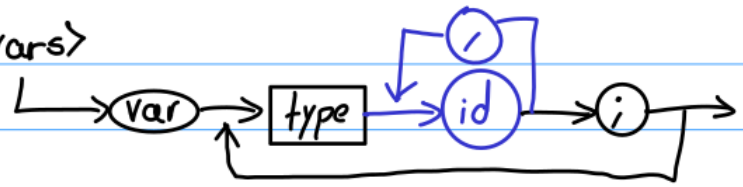
<Programa>

<Programa> → program id ; <Opt_vars> <Opt_funcs> main <Body> end

<Opt_vars> → <Vars> | ε

<Opt_funcs> → <FUNCS> | ε

<vars>



<vars> → var <DEC_VAR>

<DEC_VAR> → <TYPE> <LIST_id> ; <Dv'>

puede tener 1 o más ids

puede tener un ciclo o no

<Dv'> → <DEC_VAR> | ε

<FUNCS>

<FUNCS> → <FUNC_DEF>

<FUNC_DEF> → <RETURN_TYPE> id (<PARAMS>) <OPTIONAL_LOCAL_VARS>
<Body> ;

<RETURN_TYPE> → void | <TYPE>

<PARAMS> → <PARAM_LIST> | ε

<PARAM_LIST> → id : <TYPE> <MORE_PARAMS>

<MORE_PARAMS> → , <PARAM_LIST) | ε

<OPTIONAL_LOCAL_VARS> → <VARS> | ε

<Body>

<Body> → { <STATEMENT_LIST> }

<STATEMENT_LIST> → <STATEMENT> <STATEMENT_LIST> | ε

<TYPE>

<TYPE> → int | float

<STATEMENT>

Esta regla presenta una ambigüedad en <ASSIGN> y <F_call> ya que las 2 invocan un id, el primero es para asignar variables y el segundo es para funciones, se resolvió con el factor común izquierdo

<STATEMENT> → id <ID_STATEMENT_CONTINUATION> | <CONDITION>
| <CYCLE> | <Print>

<ID_STATEMENT_CONTINUATION> → = <EXPRESION> ; | (<EXPRESION>) ;

<ASSIGN>
<ASSIGN> → id = <EXPRESION> ;

<CONDITION>
<CONDITION> → if (<EXPRESION>) <Body> <OPTIONAL_ELSE>;
<OPTIONAL_ELSE>; → else <Body> | ε

<CYCLE>
<CYCLE> → while (<EXPRESION>) do <Body>

<F_Call >
<F_Call> → id (<EXPRESION>) ;

<Print>
<Print> → print (<PRINT_LIST>) ;
<PRINT_LIST> → <PRINT_ITEM><PRINT_MORE_ITEMS>
<PRINT_ITEM> → <EXPRESION> | cte.string
<PRINT_MORE_ITEMS> → , <PRINT_ITEM><PRINT_MORE_ITEMS> | ε

<EXPRESSION>
<EXPRESSION> → <EXP><OPTIONAL_COMPARISON>
<OPTIONAL_COMPARISON> → <RELATIONAL_OPERATOR><EXP> | ε
<RELATIONAL_OPERATOR> → > | < | !=

<EXP>

<EXP> → <TERMINO> <EXP_PRIME>

<EXP_PRIME> → + <TERMINO> <EXP_PRIME> | - <TERMINO> <EXP_PRIME>
| ε

<TERMINO>

<TERMINO> → <FACTOR> <PRIME_TERM>

<PRIME_TERM> → * <FACTOR> <PRIME_TERM>
| / <FACTOR> <PRIME_TERM> | ε

<FACTOR>

<FACTOR> → (<EXPRESION>) | id | <CTE>

<CTE>

<CTE> → cte_int | cte_float

Etapa 1

Desarrollo del Scanner y Parser

Desarrollo

Se usó Python con ANTLR para el desarrollo del compilador de BabyDuck. Los tokens y las reglas de gramática se declaran en el archivo principal con extensión **.g4.** Por convención, el nombre de los tokens tiene que ser en mayúsculas mientras que los nombres de las reglas deben empezar en minúsculas, ANTLR sigue una convención opuesta a lo que se enseñó en clases.

En clases se mostró que la forma de representar un camino vacío es con un ε y así se crearon las gramáticas libres de contexto a mano, pero fueron transformadas ligeramente al momento de ingresarlas a ANTLR, todas las reglas que representaban un elemento opcional o una repetición se plasmaron con los símbolos de ANTLR para simplificar la lectura de las reglas, se ignoraron las reglas ..._OPCIONAL y ..._PRIMA de la primera versión de la gramática libre de contexto por los símbolos *****, **+** y **?** de las expresiones regulares que son soportados por ANTLR.

De momento los tokens y gramáticas se encuentran en el mismo archivo ya que ANTLR solo lee un archivo principal. No se tiene contemplado separar el contenido en distintos archivos debido a que se implementó toda la lógica del lexer y parser en un archivo sencillo de leer. Todas las etapas que siguen son con respecto al análisis semántico y tabla de símbolos y la síntesis (backend), eso ya está afuera del alcance de ANTLR y requiere programación manual en Python

Código del lenguaje

Definición de símbolos terminales

```
53   // Palabras reservadas
54   PROGRAMA: 'programa';
55   INICIO: 'inicio';
56   FIN: 'fin';
57   VARS: 'vars';
58   ENTERO: 'entero';
59   FLOTANTE: 'flotante';
60   SI: 'si';
61   SINO: 'sino';
62   ESCRIBE: 'escribe';
63   MIENTRAS: 'mientras';
64   HAZ: 'haz';
65   NULA: 'nula';
```

```
// Operadores y puntuaciones
PARENTESIS_IZQUIERDO: '(';
PARENTESIS_DERECHO: ')';
CORCHETE_IZQUIERDO: '[';
CORCHETE_DERECHO: ']';
LLAVE_IZQUIERDA: '{';
LLAVE_DERECHA: '}';
PUNTO_Y_COMA: ';';
DOS_PUNTOS: ':';
COMA: ',';
ASIGNACION: '=';
MAS: '+';
MENOS: '-';
MULTIPLICACION: '*';
DIVISION: '/';
MAYOR_QUE: '>';
MENOR_QUE: '<';
DIFERENTE_DE: '!=';
IGUAL_QUE: '==';
```

Definición de expresiones regulares

```
// IDs y constantes
ID: [a-zA-Z_] [a-zA-Z_0-9]*;
CTE_ENT: [0-9]+;
CTE_FLOT: [0-9]+ '.' [0-9]+;
LETRERO: '"' .*? '"';

// Ignorar espacios en blanco y saltos de línea
ESPACIOS: [ \t\r\n]+ -> skip;
```

Definición de reglas

```
grammar BabyDuck;

programa: PROGRAMA ID PUNTO_Y_COMA vars? funcs* INICIO cuerpo FIN ;

vars: VARS declarar_variables+ ;
declarar_variables: declarar_ids DOS_PUNTOS tipo PUNTO_Y_COMA ;
declarar_ids: ID (COMA ID)* ;

funcs: (NULA | tipo) ID PARENTESIS_IZQUIERDO parametros? PARENTESIS_DERECHO LLAVE_IZQUIERDA vars? cuerpo LLAVE_DERECHA PUNTO_Y_COMA ;
parametros: ID DOS_PUNTOS tipo (COMA ID DOS_PUNTOS tipo)* ;

cuerpo: LLAVE_IZQUIERDA estatuto* LLAVE_DERECHA ;

tipo: ENTERO | FLOTANTE ;

estatuto: ID continuacion_de_estatuto_id | condicion | ciclo | imprime | CORCHETE_IZQUIERDO estatuto* CORCHETE_DERECHO ;
continuacion_de_estatuto_id: ASIGNACION expresion PUNTO_Y_COMA | PARENTESIS_IZQUIERDO (expresion (COMA expresion)*)? PARENTESIS_DERECHO PUNTO_Y_COMA ;

// La regla <ASIGNA> aparece en todas las reglas como una ambigüedad, por eso su lógica se implementó
// manualmente en cada caso. Como consecuencia ninguna otra regla la llama y deja de ser útil para la
// gramática. Se eliminó pero se conservó como comentario para mostrar su lugar en el diagrama de sintaxis
// asigna: ID ASIGNACION expresion PUNTO_Y_COMA ;
```

```
condicion: SI PARENTESIS_IZQUIERDO expresion PARENTESIS_DERECHO cuerpo (SINO cuerpo)? PUNTO_Y_COMA ;

ciclo: MIENTRAS PARENTESIS_IZQUIERDO expresion PARENTESIS_DERECHO HAZ cuerpo PUNTO_Y_COMA ;

// La regla <LLAMADA> aparece en todas las reglas como una ambigüedad, por eso su lógica se implementó
// manualmente en cada caso. Como consecuencia ninguna otra regla la llama y deja de ser útil para la
// gramática. Se eliminó pero se conservó como comentario para mostrar su lugar en el diagrama de sintaxis
// llamada: ID PARENTESIS_IZQUIERDO (expresion (COMA expresion)*)? PARENTESIS_DERECHO ;

imprime: ESCRIBE PARENTESIS_IZQUIERDO imprimir_elementos PARENTESIS_DERECHO PUNTO_Y_COMA ;
imprimir_elementos: ((expresion | LETRERO) (COMA (expresion | LETRERO))*) ;

expresion: exp ((MAYOR_QUE | MENOR_QUE | DIFERENTE_DE | IGUAL_QUE) exp)? ;

exp: termino ((MAS | MENOS) termino)* ;
```

```
termino: factor ((MULTIPLICACION | DIVISION) factor)* ;

factor: PARENTESIS_IZQUIERDO expresion PARENTESIS_DERECHO
    | (MAS | MENOS)? dato_o_llamada // posible ambigüedad en id, no sabe si elegir entre `id` o `id(...)`
    ;
dato_o_llamada:
    // permite elegir entre `ID` o `ID(...)`
    ID ( PARENTESIS_IZQUIERDO (expresion (COMA expresion)*)? PARENTESIS_DERECHO )?
    | cte
    ;

cte: CTE_ENT | CTE_FLOT ;
```

Workflow

Se generó con ANTLR los archivos del lexer y parser después de haber definido los elementos principales del lenguaje, se usaron los siguientes comandos:

- Generación de lexer y parser
  - **antlr4 -Dlanguage=Python3 BabyDuck.g4** → generación de BabyDuckLexer.py, BabyDuckParser.py y otros archivos
- Correr el proyecto
  - **antlr4 BabyDuck.g4** → generar archivos de java a partir del contenido del archivo principal de ANTLR
  - **javac *.java** → compilar archivos de java a bytecode
  - **grun BabyDuck programa -tokens < HelloWorld.BabyDuck** → checar la generación de tokens de un archivo del lenguaje
  - **grun BabyDuck programa -gui < HelloWorld.BabyDuck** → visualizar árbol de sintaxis a partir de un archivo del lenguaje

Test-Plan

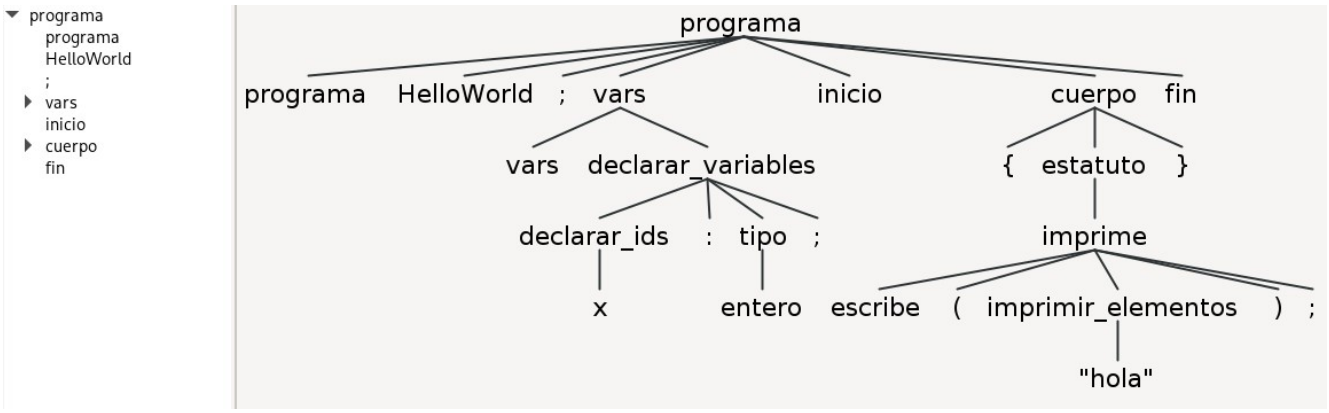
Pruebas temporales

Estas se consideran pruebas temporales debido a que son pruebas ejecutadas manualmente con el fin de demostrar la funcionalidad más básica del proyecto. Se planea usar una herramienta de automatización de pruebas como pytest para entregas a futuro que requieran una integración entre el análisis (frontend) y la síntesis (backend), no el simple funcionamiento de ambas piezas separadas.

Se realizaron 3 pruebas para esta entrega, las primeras 2 pruebas son correctas de acuerdo a la gramática pero la tercera prueba es incorrecta porque se omitió un token a propósito para probar los resultados. Los archivos de prueba se pueden encontrar en el repositorio de GitHub, aquí solo se muestran los resultados al visualizar la generación de tokens y del árbol de sintaxis.

Prueba 1

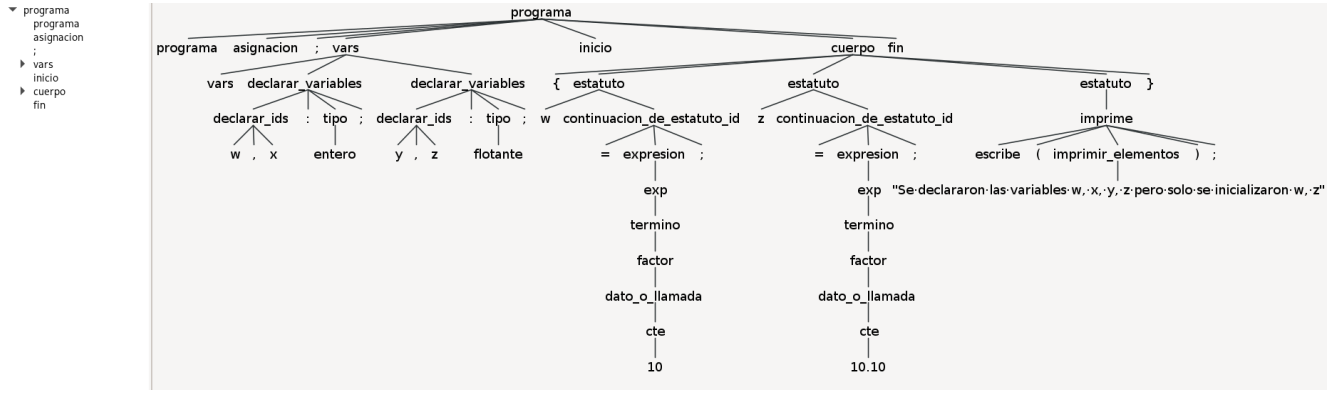
Archivo: HelloWorld.BabyDuck




```
[cocaine_ninja@nixos:~/Documents/Tec/ITC/8vo/TC3002B-Compiladores/Compiladores/Tareas/BabyDuck]$ grun BabyDuck programa -tokens < HelloWorld.BabyDuck
[@0,0:7='programa',<'programa'>,1:0]
[@1,9:18='HelloWorld',<ID>,1:9]
[@2,19:19=';',<';'>,1:19]
[@3,22:25='vars',<'vars'>,3:0]
[@4,27:27='x',<ID>,3:5]
[@5,28:28=':',<':'>,3:6]
[@6,30:35='entero',<'entero'>,3:8]
[@7,36:36=';',<';'>,3:14]
[@8,39:44='inicio',<'inicio'>,5:0]
[@9,46:46='{',<'{'>,6:0]
[@10,52:58='escribe',<'escribe'>,7:4]
[@11,59:59='(',<'('>,7:11]
[@12,60:65='"hola"',<LETRERO>,7:12]
[@13,66:66=')',<')'>,7:18]
[@14,67:67=';',<';'>,7:19]
[@15,69:69='}',<'}'>,8:0]
[@16,71:73='fin',<'fin'>,9:0]
[@17,75:74='<EOF>',<EOF>,10:0]
```

Prueba 2

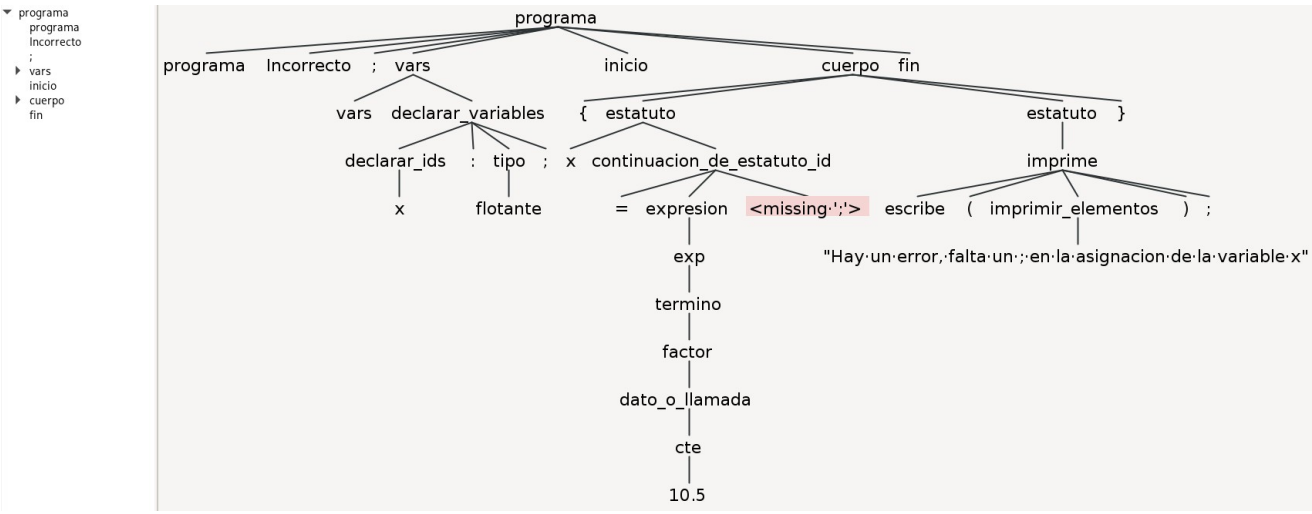
Archivo: programa2.BabyDuck




```
[cocaine_ninja@nixos:~/Documents/Tec/ITC/8vo/TC3002B-Compiladores/Compiladores/Tareas/BabyDuck]$ grun BabyDuck programa -tokens < programa2.BabyDuck
[@0,0:7='programa',<'programa'>,1:0]
[@1,9:18='asignacion',<ID>,1:9]
[@2,19:19=';',<';'>,1:19]
[@3,22:25='vars',<'vars'>,3:0]
[@4,27:27='w',<ID>,3:5]
[@5,28:28=',',<','>,3:6]
[@6,30:30='x',<ID>,3:8]
[@7,31:31=':',<':'>,3:9]
[@8,33:38='entero',<'entero'>,3:11]
[@9,39:39=';',<';'>,3:17]
[@10,41:41='y',<ID>,3:19]
[@11,42:42=',',<','>,3:20]
[@12,44:44='z',<ID>,3:22]
[@13,45:45=':',<':'>,3:23]
[@14,47:54='flotante',<'flotante'>,3:25]
[@15,55:55=';',<';'>,3:33]
[@16,58:63='inicio',<'inicio'>,5:0]
[@17,65:65='{',<'{'>,6:0]
[@18,71:71='w',<ID>,7:4]
[@19,73:73='=',<'='>,7:6]
[@20,75:76='10',<CTE_ENT>,7:8]
[@21,77:77=';',<';'>,7:10]
[@22,83:83='z',<ID>,8:4]
[@23,85:85='=',<'='>,8:6]
[@24,87:91='10.10',<CTE_FLOT>,8:8]
[@25,92:92=';',<';'>,8:13]
[@26,98:104='escribe',<'escribe'>,9:4]
[@27,105:105='(',<'('>,9:11]
[@28,106:177='"Se declararon las variables w, x, y, z pero solo se inicializaron w, z"',<LETRERO>,9:12]
[@29,178:178=')',<')'>,9:84]
[@30,179:179=';',<';'>,9:85]
[@31,181:181='}',<'}'>,10:0]
[@32,183:185='fin',<'fin'>,11:0]
[@33,187:186='<EOF>',<EOF>,12:0]
```

Prueba 3

Archivo: incorrect.BabyDuck




```
[cocaine_ninja@nixos:~/Documents/Tec/ITC/8vo/TC3002B-Compiladores/Compiladores/Tareas/BabyDuck]$ grun BabyDuck programa -tokens < incorrect.BabyDuck
[@0,0:7='programa',<'programa'>,1:0]
[@1,9:18='Incorrecto',<ID>,1:9]
[@2,19:19=';',<';'>,1:19]
[@3,21:24='vars',<'vars'>,2:0]
[@4,26:26='x',<ID>,2:5]
[@5,27:27=':',<':'>,2:6]
[@6,29:36='flotante',<'flotante'>,2:8]
[@7,37:37=';',<';'>,2:16]
[@8,40:45='inicio',<'inicio'>,4:0]
[@9,47:47='{',<'{'>,5:0]
[@10,51:51='x',<ID>,6:2]
[@11,53:53='=',<'='>,6:4]
[@12,55:58='10.5',<CTE_FLOT>,6:6]
[@13,62:68='escribe',<'escribe'>,7:2]
[@14,69:69='(',<'('>,7:9]
[@15,70:129='"Hay un error, falta un ; en la asignacion de la variable x"',<LETRERO>,7:10]
[@16,130:130=')',<')'>,7:70]
[@17,131:131=';',<';'>,7:71]
[@18,133:133='}',<'}'>,8:0]
[@19,135:137='fin',<'fin'>,9:0]
[@20,139:138='<EOF>',<EOF>,10:0]
line 7:2 missing ';' at 'escribe'
```

Hallazgos

Al inicio elegí Go como el lenguaje para implementar el proyecto BabyDuck con el módulo **participle**, pero decidí cambiar a Python con **ANTLR** debido a una mala estimación de la complejidad de la síntesis (backend). Si bien el esfuerzo de programación manual es menor con participle que con ANTLR, reflexioné y me di cuenta que me tomaría más tiempo implementar el proyecto porque estaría atrapado más tiempo resolviendo bugs del lenguaje que en entender dónde se encuentra el cuello de botella del compilador, es un lenguaje que apenas empecé a aprender recientemente y no tengo un dominio de su ecosistema como administrador de paquetes, de versiones del lenguaje, uso de herramientas, etc. Elegí ANTLR porque es una herramienta muy sólida con bastante documentación, soporta múltiples lenguajes como C#, C++, JavaScript, Java y Python.

Mientras estaba implementando las reglas de gramática en ANTLR encontré ciertos tokens que no había implementado y unos cambios ligeros en el lenguaje que generan ciertas gramáticas, eso me hizo reescribir el lenguaje que generan mis reglas gramaticales a mano antes de plasmarlas en ANTLR. Durante este proceso identiffiqué una segunda ambigüedad que no se cubrió en clases:

- <ESTATUTO> : se identificó la primera ambigüedad en clases donde se usa un id en las reglas internas <ASIGNA> y <LLAMADA>, la primera es para asignación de datos a variables y la segunda para incovar la ejecución de funciones, ambas usan un id pero con distintos casos. Se resolvió con el Factor Común Izquierdo indicando explícitamente que después de recibir un id pueden venir distintos casos.

- <FACTOR> : se agregó un camino adicional al diagrama de sintaxis ya existente, dicho camino consiste únicamente en la regla <LLAMADA> que hace uso del token id, la cuestión es que otro camino de la regla actual requiere uso específico de un id. Para solucionar esto también se utilizó el Factor Común Izquierdo porque el token que sigue de un id es diferente en ambos casos.

Esto provocó que el lenguaje de las reglas <ASIGNA> y <LLAMADA> tuviera que implementarse en el Factor Común Izquierdo, y como consecuencia ninguna otra regla llamaba a estas 2 reglas, quedaron huérfanas. Se decidió eliminarlas de la gramática porque podrían confundir al lector, pero se mantuvieron en comentarios del código indicando las razones de su eliminación y que inicialmente eran reglas del diagrama de sintaxis.

Etapa 2

La etapa semántica se encarga de evaluar reglas que son gramaticalmente correctas pero que no pueden tener sentido como sumar un entero con un flotante, usar una variable sin declarar, llamar una función con diferentes parámetros que en su declaración, etc., el proceso de compilación debe detenerse por completo al encontrar un error semántico.

ANTLR genera el Abstract Syntax Tree después del proceso de parsing, es aquí donde se deben aplicar los mecanismos de evaluación semántica como puntos neurálgicos, el cubo semántico en expresiones, el directorio de funciones y tabla de variables para …….

Cubo Semántico

Esta estructura permite evaluar si una expresión es válida. En expresiones aritméticas no se puede sumar un string con un entero, pero algunos lenguajes como JavaScript si permiten estas operaciones porque así es la filosofía del lenguaje de acuerdo a los autores, así pican ellos la cebolla. Esta serie de reglas se declara en el cubo semántico, es una estructura de 3 dimensiones que muestra de forma eficiente el resultado al juntar 2 datos con una expresión específica. Este paso se puede realizar de otras formas como una serie de IF statements anidados, switch conditions, etc., funcionan pero no son la forma más eficiente.

La implementación se realizó con diccionarios anidados usando como llaves los strings en cada una de las dimensiones, en la primera dimensión se encuentra el dato izquierdo, en la segunda dimensión el dato derecho y en la tercera dimensión el operador usado, esto da un resultado que puede ser el tipo de dato permitido, una operación válida (ok) o un error (error). La complejidad temporal de búsqueda es O(1) ya que el tamaño de la estructura es finito, los diccionarios funcionan con hash tables internamente. Condensa la información y realiza búsquedas eficientes. Las reglas implementadas para el proyecto son las siguientes:

- Solo se pueden hacer operaciones con un mismo tipo de dato, no se puede mezclar enteros con flotantes
- El booleano no es un tipo de variables que se pueda usar, pero se incluyó en el cubo porque esta representación existe en las expresiones y también en evaluaciones con los operadores **==** y **!=**
- El string tampoco es un tipo de dato que se pueda declarar, a diferencia del booleano, no se incluyó en el cubo semántico porque no se puede hacer ninguna operación con los strings, solamente se puede imprimir. Cualquier tipo de dato que no se encuentre en el cubo semántico en una expresión es una operación inválida

```
semantic_cube = {
    "entero": {
        "entero": {
            "+": "entero",
            "-": "entero",
            "*": "entero",
            "/": "entero",
            ">": "booleano",
            "<": "booleano",
            "==": "booleano",
            "!=": "booleano",
            "=": "ok"
        },
        "flotante": {
            "+": "error",
            "-": "error",
            "*": "error",
            "/": "error",
            ">": "error",
            "<": "error",
            "==": "error",
            "!=": "error",
            "=": "error"
        },
        "booleano": {
            "+": "error",
            "-": "error",
            "*": "error",
            "/": "error",
            ">": "error",
            "<": "error",
            "==": "error",
            "!=": "error",
            "=": "error"
        }
    },
    "flotante": {
        "entero": {
            "+": "error",
            "-": "error",
            "*": "error",
            "/": "error",
            ">": "error"
```

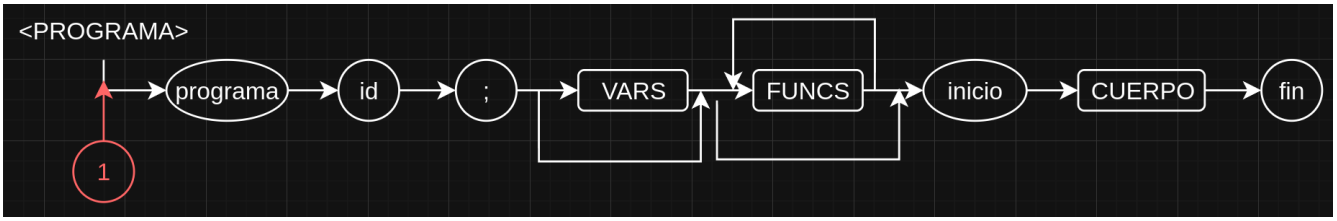
```
semantic_cube = {
    "entero": {
        "entero": {…
        },
        "flotante": {…
        },
        "booleano": {…
        }
    },
    "flotante": {
        "entero": {…
        },
        "flotante": {…
        },
        "booleano": {…
        }
    },
    "booleano": {
        "entero": {…
        },
        "flotante": {…
        },
        "booleano": {…
        }
    }
}
```

Puntos Neurálgicos

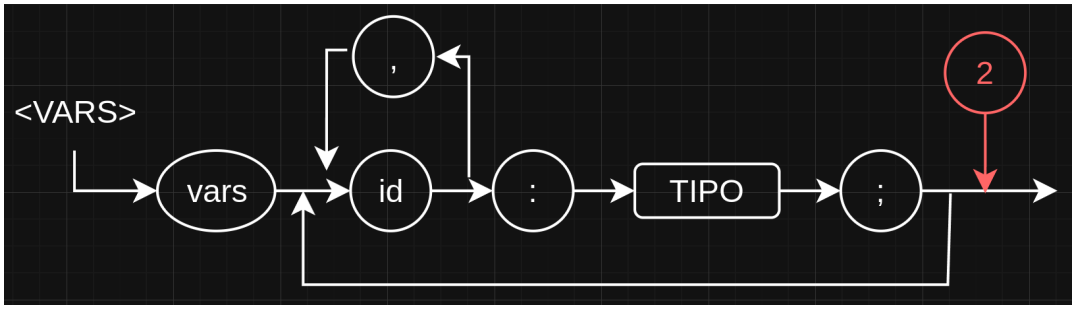
Son necesarios para validar la semántica de reglas que son gramaticalmente correctas pero que no tienen sentido. Los cuádruplos se generan en cada punto neurálgico que se ejecuta correctamente y se detiene el programa cuando se encuentra algo que no tiene sentido en el programa.

Se utilizó una clase que hereda de **BabyDuckVisitor** para implementar los puntos neurálgicos como métodos de la clase. En clases se mostró que se pueden agregar varios puntos neurálgicos en una misma regla, pero en este proyecto se agruparon todas las lógicas posibles en un solo punto neurálgico por regla debido a que al declarar un punto neurálgico en ANTLR se tiene que declarar toda la lógica de las sub-reglas en el punto neurálgico, ANTLR te da control absoluto en la regla de modo que omitirá el análisis de sub-reglas no especificadas en el punto neurálgico aunque si estén definidas en la gramática, por eso se agruparon varias tareas en un solo punto neurálgico a excepción de aquellas que se

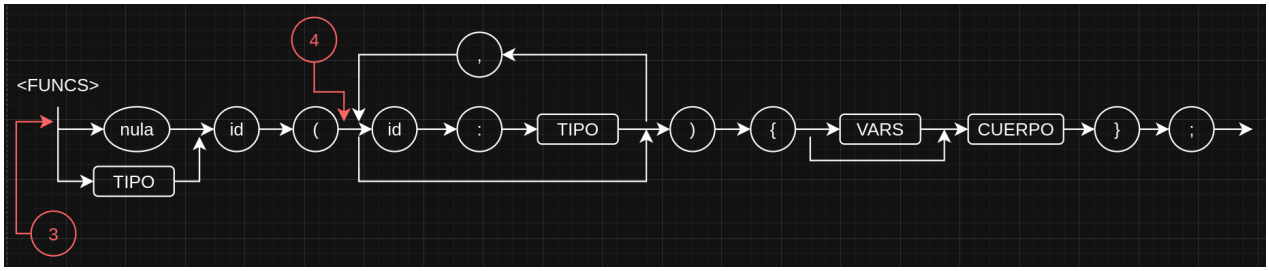
representen con reglas derivadas de reglas más complejas como en **<FUNCS>**, se derivó la regla **<PARAMETROS>**, este es un segundo punto neurálgico de **<FUNCS>** que verifique la declaración correcta de los parámetros. <FUNCS> requiere la declaración de un punto neurálgico y la visita manual de la regla derivada **<PARAMETROS>**, y esta a su vez contiene un segundo punto neurálgico.



- Inicializar el directorio de funciones y crear el scope global cada vez que se inicia un programa



- Agregar variables al scope actual y verficar que no se declare una variable duplicada



Directorio de Funciones y Tablas de Variables

El directorio de funciones y las tablas de variables son la memoria del compilador. Su trabajo es recordar qué variables y funciones se han declarado para poder atrapar errores o funcionar apropiadamente en los puntos neurálgicos.

El directorio de funciones es el índice principal del programa, indica qué funciones existen en total y qué tipo de valor devuelven. La estructura de datos principal es un diccionario de Python, la llave es el nombre de la función y el valor asociado es un objeto que guarda toda su información como el tipo de retorno, parámetros, etc.

La tabla de variables el inventario dentro de cada función, puesto que en este proyecto solo se maneja el scope global y local por función, no por cada estatuto. Cada función tiene su propia tabla de variables, que nos contiene únicamente las variables declaradas en ese contexto. Cada tabla de variables dentro de una función se representa también como un diccionario. La llave es el nombre de la variable y el valor asociado es un objeto que guarda su tipo.

Extras

Detalles que no son relativos a la literatura de compiladores, pero son features importantes para llevar un manejo eficiente en una codebase compleja. No es un proyecto grande, pero cada etapa del compilador funciona con una lógica diferente, cada pipeline es único y por eso es importante mantener un orden en cada aspecto del proyecto.

Inicialización del programa

Se creó un pequeño pipeline de compilación para implementar separación de intereses.

- **main.py**: archivo principal de Python que llama al pipeline de compilación y maneja errores en un alto nivel.
- **compiler.py**: contiene todo el pipeline de compilación simplificado en funciones. Llama todas las etapas del compilador: léxico, sintáctico, semántico, generación de cuádruplos y ejecución de código intermedio en la máquina virtual. Se imprimen estatutos en la terminal cada vez que se termina una etapa exitosamente para facilitar debugging.

Manejo de errores a nivel global

Los errores del lexer y parser son detectados por ANTLR y detecta cuando se usa un token que no fue declarado en el archivo principal o cuando no se siguen reglas de gramática correctamente, pero ya no es tarea de ANTLR detectar errores en la etapa semántica ni en la ejecución de código intermedio. No se implementó en la generación de cuádruplos porque cada cuádruplo se genera al terminar un punto neurálgico exitosamente. Por eso se implementó una clase de Python dedicada al manejo de errores en cada etapa. Imprime en grande el título de la etapa en donde se detectó el error y también en la subsección correspondiente. Esto permite debuggear con mayor facilidad durante la etapa de desarrollo

cuando surgen bugs en la lógica del código y cuando se ingresa un archivo semánticamente incorrecto.

Los errores se manejan en los puntos neurálgicos cuando puede suceder un error, pero también se tienen que manejar en la ejecución de código intermedio porque hay tareas que corresponden al momento de ejecutar expresiones como una división entre cero o cuando se supera el límite de llamadas recursivas (un StackOverflow).

Etapa 3

No se implementaron todas las funcionalidades de la Etapa 3 debido a la falta de tiempo, a continuación se describen los avances realizados en el código.

Se implementaron todas las estructuras de datos necesarias para la generación de cuádruplos, pero aún falta integrar su actividad en los puntos neurálgicos.

- **self.quadruples**: la fila de cuádruplos, es una lista de Python que funciona conceptualmente como una fila. Almacena la secuencia completa de cuádruplos generados durante el análisis. Cada cuádruplo añadido representa una instrucción del código intermedio.
- **self.operand_stack**: la pila de operandos, almacena temporalmente los operandos a medida que el compilador los encuentra al analizar una expresión. Los operandos pueden ser constantes, nombres de variables o los nombres de variables temporales.
- **self.type_stack**: la pila de tipos que trabaja en paralelo y sincronía con la operand_stack. Por cada operando que se ingresa a la operand_stack, su tipo de dato correspondiente (“int”, “float”, “bool”) se ingresa en esta type_stack
- **self.operator_stack**: la pila de operadores implementada como lista para gestionar el orden y precedencia de operadores pendientes de ser aplicados.
- **self.jump_stack**: la pila de saltos implementada como lista para la traducción de estatutos de control como “si” y “mientras”, no guarda operadores ni operandos, solo los índices de los cuádruplos de salto.
- **self.temp_counter**: atributo de tipo entero que actúa como un contador secuencial para generar nombres únicos de variables temporales (t1, t2, t3, …, tn)

- **self.pending_jump**: atributo que almacena un valor constante, npo ues una estructura de datos, sino un marcador de posición para rellenar el campo de destino de un cuádruplo de salto cuando este se genera.

También se implementó en código una helper function para crear el cuádruplo de una expresión, agregarlo a la fila y devolver su índice. Aún falta la implementación de estatutos lineales.

Uso de inteligencia artificial

En este trabajo se usó inteligencia artificial de manera responsable para automatizar tareas manuales y repetitivas, no para hacer todo el trabajo con pocos prompts. Se usó ChatGPT 4.1-mini para las siguientes tareas:

- Generar archivos de prueba correctos e incorrectos, son los archivos que se encuentran en **BabyDuck/Tests**
- Generar el cubo semántico: esta fue una tarea bastante repetitiva, se usó IA para generar la estructura y yo asigné las reglas que quiero para el programa: solo se pueden hacer operaciones aritméticas y relacionales con un mismo tipo de dato (int con int, float con float), solo se pueden hacer comparaciones con operadores booleanos y los strings no son un tipo de dato, solo son para imprimir en la terminal
- Ayuda rápida en sintaxis muy específica ANTLR:
  - Generar el lexer y parser a partir de los tokens y gramática ingresada: antlr4 -Dlanguage=Python3 BabyDuck.g4
  - Generar en una interfaz el abstract sintax tree: grun BabyDuck programa -gui < HelloWorld.BabyDuck
  - Generar en la terminal los tokens y reglas del script ingresado: grun BabyDuck programa -tokens < HelloWorld.BabyDuck
  - Sintaxis en puntos neurálgicos: visitar una regla con **ctx.visit(←rule->())** y acceder al valor de un token con **ctx.ID().getText()**