Campus Tecnológico Central de Cartago
Compiladores e Intérpretes

Etapa 2: Analizador Sintáctico

Prof. Kirstein Gätjens Soto

Daniel Sequeira Retana - 2019039641
Kenneth Palacios Molina - 2020035407

Fecha de entrega: 13 de mayo

I Semestre

2024

índice

índice 2
Propuesta: Concurso de nombres 5
Definición del lenguaje 6
Autómata 11
Autómata de la letra A 12
Autómata de la letra C 13
Autómata de la letra E 14
Autómata de la letra F 15
Autómata de la letra I 16
Autómata de la letra J 17
Autómata de la letra L 19
Autómata de la letra M 20
Autómata de la letra N 21
Autómata de la letra O 22
Autómata de la letra P 23
Autómata de la letra Q 24
Autómata de la letra S 25
Autómata de la letra T 26
Autómata de la letra Y 27
Índice de pruebas 28
Propuesta gramatical 28
Estructura del título del programa nhequéquequé 28
Apertura de sección de constantes octará 28
Apertura de sección de tipos aíca 29
Apertura de sección de variables tilhtilh 29
Apertura de sección de rutinas purú 29
Apertura de sección de prototipos chá 29
Sistema de asignación de constantes arrá 30
Sistema de asignación de tipos putú 30
Sistema de asignación de variables nicó 30
Sistema de asignación de booleanos afá 31
Sistema de asignación de registros poréteca … turé 31
Tipos de datos 32
Literales de conjuntos 32
Literales de archivos 32
Literales de números flotantes 33
Literales de enteros 33
Literales de caracteres 34
Literales de strings 35
Literales de arreglos 35

Literales de registros....36
Sistema de acceso a arreglos....36
Sistema de acceso a strings....36
Sistema de acceso a registros....36
Asignación con: := += -= *= %= /=....37
Operaciones aritméticas básicas de enteros: + - * % /....37
Decremento e incremento de variables: jené y taqué....38
Operaciones básicas sobre caracteres: >> << &? #?....38
Operaciones lógicas: ta yú pucá emé....39
Operaciones de strings: Concatenar: catáqui / Length: lharátesufá / cortar: joné / recortar: nhalh / encontrar: cochéqui....40
Operaciones con conjuntos: Agregar: erronh / Borrar:quené / Union: fuinhye / Interseccion: sulí / PerteneceA: pcalúfa / Vacío?: culíte....42
Operaciones con archivos: Abrir: quijí / Cerrar: eloc / Escribir: lhioc / Leer: lhaic / Crear: quijérri / Asociar: láte....43
Operaciones con números flotantes:|++ ++| |+| +|+....43
Operaciones de comparación: < > = >= <= ><....44
Manejo de bloques de más de una instrucción: cojcoj … fuchí....45
Instrucción while....45
Instrucción if....46
Instrucción if-then-else....46
Instrucción switch....46
Instrucción repeat-until....47
Instrucción for....47
Instrucciones break, continue y halt....47
Encabezado de funciones....48
Encabezado de procedimientos....48
Manejo de parámetros formales....48
Manejo de parámetros reales....48
Instrucción return....48
Operación sizeof....48
Sistema de coerción de tipos....48
Salida Estándar....49
Entrada Estándar....49
Pruebas de análisis léxico....49
Test de comentarios....49
Test de Seccion de constantes....49
Pruebas de análisis sintáctico....50
Test de comentarios....50
Test de fors....50
Test de ifs....50
Test de inicio incorrecto....50
Test de repeat until....51
Test de sección de constantes....51
Test de falta de inicio....51

Test de sección de prototipos.....51
Test de sección de rutinas.....51
Test de sección de tipos.....51
Test de sección de variables.....51
Test de while.....52
Funcionamiento del scanner.....52
Funcionamiento del parser.....53
Tabla de símbolos.....60

Propuesta: Concurso de nombres

Nombre propuesto: Táfac

Significado: Diablo celestial de los que tienen como función dar a los condenados su dosis diaria de tormento, que consiste en que los mastican, se los tragan y luego los expulsan, convertidos en bagazo, al defecar, orinar o vomitar

Logotipo propuesto:

Extensión para los archivos propuesta: .taf

Definición del lenguaje

Estructura o parte del lenguaje	Elemento en el lenguaje	Traducciones del Malecu	Comentarios / Aclaraciones
Estructura del título del programa FC	nhequéquequé	Sonido onomatopéyico del chirrido de una puerta.	
Apertura sección constantes	octará	Traducción de piedra	escrito en piedra
Apertura sección de tipos	aíca	Traducción de varios	tipos equivale a que hay variedad
Apertura de la sección de variables	tí	Traducción de agua, en contraste a piedra	
Apertura de la sección de rutinas	urúma	Traducción de costumbre	
Apertura de la sección de prototipos	chóje	Traducción de proteger, un prototipo protege de un error declarando la función antes de su definición	
Punto de entrada del programa	cojcoj	Tocar la puerta para poder ingresar	
Sistema de asignación de constantes	caralh	Sonido de una piedra chocando con otra	
Sistema de asignación de tipos	putú	Traducción de cielo	Hay muchos tipos de cielo
Sistema de declaración de variables	sulí	Traducción de Cambio	
Tipo de dato entero	chátené	Entero o completo	
Tipo de dato carácter	filhílhi	Representación	

Tipo de dato string	purucpuruc	Cuerda	
Tipo de dato booleano	cochója	Doblado	
Tipo de dato conjunto	apúchá	Conjunto	
Tipo de dato archivo de texto	puríri	Contener	
Tipo de datos números flotantes	jané	Flotar	
Tipo de dato arreglos	cóye	Guardado/ Tener guardado	Los arreglos son sin límite de dimensión y son homogéneos y de cualquier tipo del lenguaje
Tipo de dato registros	lhará	Seña	
Tipo de datos atómico creativo	Número imaginario: úji	Traducción de imaginarse	
Literales booleanas	True: suírra / False: epéme	Traducción literal de verdadero y parcial de falso	
Literales de conjuntos	{ 'X1', 'X2',….,'Xn' }		Usa las literales de caracteres, separados por comas
Literales de archivos	{@ "c:\akw\tarea1.txt" , L @}		La primer literal de string es la ruta y nombre con extensión del archivo físico. La letra indica si es de lectura (L), escritura (S), ambas (D)
Literales de números flotantes	1.223		
Literales de enteros	-123 0xF4EC		Las de C, exactamente las de C.
Literales de caracteres	Con comilla simple: 'A' '\n' '\666'		Las de C, exactamente las de C.

Literales de strings	"Hola \n"		Las de C, exactamente las de C.
Literales del tipo de datos atómico creativo	3i un numero entero finalizando con una i		
Literales de arreglos	<[1,2,3,4]>		Llaves con corchetes
Literales de registros	úpalá [[ ]]	Traducción de casa	Brackets dobles
Sistema de acceso arreglos	<[]>		
Sistema de acceso strings	¿?		
Sistema de acceso registros	-->		
Asignación y Familia	óri / <operación>óri	Dar órdenes a alguien	Escoger un símbolo o palabra para la asignación y una sintaxis para las asignaciones combinadas.
Operaciones aritméticas básicas de enteros	+ - * % /		Suma Resta Multiplicación Residuo y División Entera
Incremento y Decremento	fay+ / tij-	Aumentar: fayátaqué / Disminuir: tíjye	Solo en prefijo
2 operaciones de enteros creativas	curíjurí para logaritmo / caquí: suma del piso x en el triángulo de pascal	Traducción de ritual / traducción de pilar	
Operaciones básicas sobre caracteres	Pasar a mayúsculas: taq / Pasar a minúsculas: jen / is alpha: mal / is digit: jat	Subir: taqué / Bajar: jené / Letra: malhióca / Numeroso: jatájatá	Pasar a mayúsculas, a minúsculas, is alpha, is digit, todas son unarias y en prefijo.
Operaciones lógicas solicitadas	and: muri / or: cóqui / xor: tiátiá / not: emé	Traducción de and y Traducción de suceder (un or es esperar que algo suceda). Xor: Traducción de	

		exactamente Not: Traducción de no	
Operaciones de Strings solicitadas	Concatenar: catáqui / Length: lharátesufá / cortar: joné / recortar: nhalh / encontrar: cochéqui	Adherir: catáqui / Cosa larga: lharátesufá / Cortar: joné / Ideófono del sonido del cortar de un solo golpe un objeto: nhalh / Encontrar: cochéqui	Concatenar length cortar recortar find
Operaciones de conjuntos solicitadas	Agregar: erronh / Borrar: quené / Union: fuinhye / Interseccion: sulí / PerteneceA: pcalúfa / Vacío?: culíte	Añadir: erronh / Borrar: quené / Mezclar: fuinhye / Cruzar: sulí / Miembro: pcalúfa / Ser vaciado: culíte	Agregar Borrar Union Interseccion PerteneceA Vacío?
Operaciones de archivos solicitadas	Abrir: quijí / Cerrar: eloc / Escribir: lhioc / Leer: lhaic / Crear: quijérri / Asociar: láte	Abrir: quijí / Cerrar: eloc / Escribir: lhioc / Leer: lhaic / Crear: quijérri / Relación: láte	abrir cerrar escribir leer crear asociar
Operaciones de números flotantes	Suma: erroc / Resta: sírru / Multiplicación-> fují / División: palé	Añadir: erroc / Quitar: sírru / Acumulación: fují / Dividir: palé	suma, resta, multiplicación, división (binarias)
Operaciones del tipo de datos atómico creativo	i+ i* i/ i-		Suma, multiplicación, división, resta
Operaciones de comparación solicitadas	> < = >= <= ><		Se comparan enteros y retornan booleanos
Manejo de Bloques de más de una instrucción	CojCoj		Hay que pensar el de cierre. O sugerir un par diferentes si encuentran algo más divertido.
Instrucción while	pují <exp> sóta / pují (<exp>)	Dedicar el tiempo a algo: pují / Hacer: sóta	while <exp> do while (<exp>)
Instrucción if-then-else	if: arú / else: ótacá		Como el ifTrue de Smalltalk!! then y else opcionales y en cualquier orden.
Instrucción switch	Switch: lhifíji / case: palhá / default: quírra	Cambiar: lhifíji / Indica que llego el momento de hacer algo: palhá / Origen: quírra	switch case default Como el de C.

Instrucción Repeat-until	paípaí / yája	Repetidamente: paípaí / Hasta: yája	como el repeat until de pascal.
Instrucción For	ajá	Para: ajá	como el for de pascal no de C puede ser como el de python
Instrucción With	tiní	Con: tiní	como el whith de pascal
2 Instrucciones creativas	Tarráni: hacer una pausa en la ejecución hasta que se reciba una entrada en el teclado / Chequéle: equivalente a un cls, limpiar consola	Traducción de tiempo	
Instrucción break	foróforóye	Romperle la entrada de la cueva a una taltuza	como el de C
Instrucción continue	taráne	Seguir la línea de algo como se determinó previamente	como el de C
Instrucción Halt	lauc	Terminar	Detiene la ejecución del programa completamente.
Encabezado de funciones	lufá id (...) : tipo	Actividad	buscar palabra melecu
Encabezado de procedimientos	ónhaónha id (...)	Conducta	buscar palabra melecu
Manejo de parámetros formales	x(.a, .b, .c)		Deben ser consistentes con la declaración de variables. Hay que pensar la forma de decir que un parámetro va por referencia y no por valor.
Manejo de parámetros reales	x(a, b, c)		
Instrucción return	jalac	Devolver	Hay dos trabajos diferentes que hace el return de C. ¿Los separamos? Sí, proponemos que

			jalac no cumpla la función de terminar el programa
Operación de size of	níca	De este tamaño	tamaño ya sea de un tipo o de una expresión.
Sistema de coerción de tipos	ü(nombre_del_tipo)		expresión (pensar un operador en malecu) tipo
Manejo de la entrada estándar	tioctipo	Entrar: tioc	Los seis tipos atómicos
Manejo de la salida estándar	pítetipo	Salir: píte	Los seis tipos atómicos
Terminador	;		Es el punto y coma y es un terminador

Autómata

El autómata del proyecto fue desarrollado por partes debido a su alta complejidad. Aún así, se hizo de manera que sea simple de entender y cada “diagrama individual” comienza en el estado q0 y tiene conexiones hacia otros estados dependiendo del token que esté formando. Los caminos no definidos para cada estado se enciclan, es decir, no son aceptados por el autómata.

Autómata de la letra A

Autómata de la letra C

Autómata de la letra E

Autómata de la letra F

Autómata de la letra l

Autómata de la letra J

Autómata de la letra L

193%

Autómata de la letra M

Autómata de la letra N

Autómata de la letra O

Autómata de la letra P

Autómata de la letra Q

Autómata de la letra S

Autómata de la letra T

t
u
o
ó
i
r
r
r
r
c
n
í
q
u
u
é
e
í
e
t
e
i
r
á
á
l
h
t
i
l
e
a
h
l
e
e
a
f
l
e
q
u
é
q
r
l
e
l
h
t
i
q
u
é
e
e
e

Autómata de la letra Y

Índice de pruebas

Asignaciones.txt

- Contiene ejemplos de las asignaciones

instrucciones.txt

- Contiene ejemplos de las instrucciones

Operaciones.txt

- Contiene ejemplos de las instrucciones

Tipos-de-datos.txt

- Contiene ejemplos de los tipos de datos

Propuesta gramatical

Estructura del título del programa nhequéquequé

Código:

```
## esto va al inicio del programa
## y declara el nombre del mismo
nhequéquequé miPrograma;
```

<apertura> ::= nhequéquequé id ;

Apertura de sección de constantes octará

En código:

```
##apertura de sección de constantes: octará
octará

    ## arrá instrucción prefija que declara x1
    arrá x1 45;
    arrá x2 'd';
    arrá x3 maíca; ##maica es False
```

<const> ::= octará <expresión>

<expresión> ::= ... <expresión>

<expresión> ::= ε

Apertura de sección de tipos aíca

En código:

```
##apertura de sección de tipos: aíca
aíca

    ## putú es una instrucción prefija
    putú Edad chátené;
```

<tipoSec> ::= aíca <expresión>

Apertura de sección de variables tilhtilh

En código:

```
##apertura de sección de variables: tilhtilh
tilhtilh

    ##chátené es tipo entero y nicó su asignación,
    chátené x4 nicó 23;
    coyé chátené x5 ta 10;                ## esto de
    ílo x6 nicó "Esto es un string";      ## ílo es
```

<variableSec> ::= tilhtilh <expresión>

Apertura de sección de rutinas purú

En código:

```
## sección de rutinas (funciones y procedimientos): purú
purú

macorróca afá esUnaVocal (malhióca Letra)
cojcoj                    ## se puede interpretar como {
```

<rutinasSec> ::= purú <expresión>

Apertura de sección de prototipos chá

En código:

```
## sección de prototipos (se refiere a la declar
chá
    ##macorróca es el encabezado de la función,
    macorróca chátené fibonacci (chátené N);
    óri factorial (chátené N, suí chátené Resp);
    macorróca afá esUnaVocal (malhióca Letra);
```

<prototiposSec> ::= chá <expresion>

Sistema de asignación de constantes arrá

En código:

```
## arrá instrucción prefija que
arrá x1 45;
arrá x2 'd';
arrá x3 maíca; ##maica es False
```

<asigConst> ::= arrá id <literalX> ;

Sistema de asignación de tipos putú

En código:

```
## putú es una instr
putú Edad chátené;
```

<asigTipo> ::= putú id <tipoDato> ;

Sistema de asignación de variables nicó

En código:

```
##chátené es tipo entero y nicó su
chátené x4 nicó 23;
coyé chátené x5 ta 10;
ílo x6 nicó "Esto es un string";
afá x7 nicó tócu;
```

<asigVar>::= <tiposDeDatos> id nicó <literalX> ;

Sistema de asignación de booleanos afá

En código:

```
afá x7 nicó tócu;
```

<booleanoTipo> ::= afá id <asignacionX> ;

<asignacionX> ::= nicó <booleano>

<asignacionX> ::= ε

<booleano> ::= tócu

<booleano> ::= maíca

Sistema de asignación de registros poréteca … turé

En código:

```
##Esto es el equivalente
poréteca
    chátené campo1;
    malhióca campo2;
    ílo campo3;
    afá campo4;
turé X nicó .... ;
```

<reg> ::= poréteca <listaDatos> turé id <asignacionX> ;

<listaDatos> ::= <nuevoDato>

<listaDatos> ::= <nuevoDato> <listaDatos>

<asignacionX> ::= ε

Tipos de datos

En código:

```
chátené campo1;
malhióca campo2;
ílo campo3;
afá campo4;
```

<tiposDeDatos> ::= <tipoDato> id <asignacionX> ;

<asignacionX> ::= ε

<tipoDato> ::= chátené

<tipoDato> ::= malhióca

<tipoDato> ::= ílo

<tipoDato> ::= afá

<tipoDato> ::= apúchá

<tipoDato> ::= Ihalámi

<tipoDato> ::= jané

<tipoDato> ::= coyé <tipoDato> id ta

Literales de conjuntos

En código:

```
apúchá x8 nicó {x4,x5,x6,x7};
```

<S> ::= apuchá id <asignacionX> ;

<asignacionX> ::= nicó { <conjunto> } | ε

<conjunto> ::= id

<conjunto> ::= id, <conjunto>

Literales de archivos

En código:

```
lhalámi x9 nicó {{ "c:\akw\tarea1.txt" , L }} ;
```

<S> ::= lhalámi id <asignacionX> ;

<asignacionX> ::= nicó {{ <stringRuta>, <accionArchivo> }} | ε

<accionArchivo> ::= L

<accionArchivo> ::= S

<accionArchivo> ::= D

Literales de números flotantes

En código:

```
jané x10 nicó 1.89;
```

Es importante resaltar que los literales de los flotantes en Táfac son iguales a los del lenguaje C, por lo que la gramática de C influenció fuertemente nuestra propuesta.

<S> ::= jané id <asignacionX> ;

<asignacionX> ::= nicó <flotante> | ε

<flotante> ::= <entero>. <parte_fraccionaria> <exponente>

<exponente> ::= E + <entero>

<exponente> ::= E - <entero>

<exponente> ::= E <entero>

<exponente> ::= €

<entero> ::= <digito>

<entero> ::= <digito> <entero>

<entero> ::= €

<parte_fraccionaria> ::= <digito>

<parte_fraccionaria> ::=< digito> <parte_fraccionaria>

<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Literales de enteros

En código:

```
chátené x4 nicó 23;
```

Es importante resaltar que los literales de los enteros en Táfac son iguales a los del lenguaje C, por lo que la gramática de C influenció fuertemente nuestra propuesta.

<S> ::= chátené id <asignacionx> ;

<asignacionX> ::= nicó <entero> | ε

<entero> ::=< entero_decimal>

<entero> ::=< entero_octal>

<entero> ::=< entero_hexadecimal>

<entero_decimal> ::= <digito_no_cero>

<entero_decimal> ::=< digito_no_cero><digitos_decimales>

<entero_decimal> ::= 0

<entero_octal> ::= 0

<entero_octal> ::= 0 <digitos_octales>

<entero_hexadecimal> ::= 0 X<digitos_hexadecimales>

<digitos_decimales> ::=< digito_decimal>

<digitos_decimales> ::=< digito_decimal><digitos_decimales>

<digitos_octales> ::=< digito_octal>

<digitos_octales> ::=< digito_octal><digitos_octales>

<digitos_hexadecimales> ::= <digito_hexadecimal>

<digitos_hexadecimales> ::=< digito_hexadecimal><digitos_hexadecimales>

<digito_no_cero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digito_decimal> ::= 0 | <digito_no_cero>

<digito_octal> ::= 0 | 1 | 2 |3|4|5|6|7

<digito_hexadecimal> ::=< digito_decimal> | A | B | C | D | E|F

Literales de caracteres

En código:

```
malhióca x11 nicó ':';
```

Es importante resaltar que los literales de los caracteres en Táfac son iguales a los del lenguaje C, por lo que la gramática de C influenció fuertemente nuestra propuesta. Se evita

escribir todos los posibles caracteres por comodidad visual, pero se entiende que son los mismos que se aceptarían en C.

<S> ::= malhióca id <asignacionX> ;

<asignacionX> ::= nicó <caracter> | ε

<caracter> ::= '<caracter_literal>'

<caracter> ::= '<escape_secuencia>'

<caracter_literal> ::= cualquier carácter visible excepto' y

<escape_secuencia> ::=< combinacionesValidasC>

Literales de strings

En código:

```
ílo x6 nicó "Esto es un string";
```

Es importante resaltar que los literales de los strings en Táfac son iguales a los del lenguaje C, por lo que la gramática de C influenció fuertemente nuestra propuesta. Se evita escribir todos los posibles caracteres por comodidad visual, pero se entiende que son los mismos que se aceptarían en C.

<S> ::= ílo id <asignacionX> ;

<asignacionX> ::= nicó <string> | ε

<string> ::= "<string_literal> <escape_secuencia>"

<string_literal> ::= <combinacionesStringsC>

<string_literal> ::= €

<escape_secuencia> ::=< combinacionesValidasC>

<escape_secuencia> ::= €

Literales de arreglos

En código:

```
coyé chátené x12 nicó {[ 1,2,3,4 ]};
```

<S> ::= coyé <tipoDato> id <asignacionX> ;

<asignacionX> ::= nicó <arreglo> | ε

<arreglo> ::= { [<datosDeTipoDato>] }

<datosDeTipoDato> ::= <dato>

<datosDeTipoDato> ::=< dato>, <datosDeTipoDato>

<dato> ::= id

<dato> ::= <datoDeTipoDato>

Literales de registros

En código:

```
porétecá                                    ## con
    chátené campo1;                         ## Los
    malhióca campo2;                        ## lue
    ilo campo3;
    afá campo4;
turé X nicó {@ 1, 'a', "string", x7 @} ;
```

<S> ::= porétecá <listaDatos> turé id <asignacionX> ;

<asignacionX> ::= nicó <registro> | ε

<registro> ::= {@ <datosCorrespondientes> @ }

<datosCorrespondientes> ::= <dato>

<datosCorrespondientes> ::= <dato>, <datosDeTipoDato>

<dato> ::= id

<dato> ::= <datoCorrespondiente>

Sistema de acceso a arreglos

En código:

```
x5[1] := 6;
```

<S> ::= id [<accesador>] :=< dato> ;

<accesador> ::= id

<accesador> ::= <entero>

Sistema de acceso a strings

En código:

```
x6 $$ 3;
```

<S> ::= id $$ <entero> ;

Sistema de acceso a registros

En código:

```
campo1@A := 25;
```

<S> ::= idDato @ idRegistro :=< datoTipoDato> ;

Asignación con: := += -= *= %= /=

En código:

```
chátené x4 nicó 23;
campo1@A := 25;

x4 += 2;
x4 -= 1;
x5 *= 10;
x6 %= 2;
x7 /= 3;
```

<asiganciones> ::= idDato <asignacion> <datoTipoDato> ;

<asignacion> ::= :=

<asignacion> ::= +=

<asignacion> ::= -=

<asignacion> ::= *=

<asignacion> ::= %=

<asignacion> ::= /=

Operaciones aritméticas básicas de enteros: + - * % /

En código:

```
chátené x4 nicó 23;
chátené x5;

x5 := x4 + 5;
x5 := x4 - 3;
x5 := x4 * 4;
x5 := x4 / 2;
x5 := x4 % 6;
```

<S> ::=< dato> <operacionEntero> <dato> ;

<dato> ::= id

<dato> ::= <entero>

<operacionEntero> ::= *

<operacionEntero> ::= /

<operacionEntero> ::= %

<operacionEntero> ::= +

<operacionEntero> ::= -

Decremento e incremento de variables: jené y taqué

En código:

```
chatené x nicó 23;
x jené;
```

<DecIncremento> ::= id <cremento>

<cremento> jené

<cremento> taqué

Operaciones básicas sobre caracteres: >> << &? #?

En código:

```
##definimos el caracter
malhióca char1 nicó 'f';

##Mayuscula
>> char1;
→ 'F'

##minúscula
<< char1;
→ 'f'

##is Alpha (o sea letra)
&? char1;
→ tócu

##is digit (numero)
#? char1;
→ maíca
```

<operacionesCaracteres> ::= <operadorCaracter> id

<operadorCaracter> ::= >>

<operadorCaracter> ::=<<

<operadorCaracter> ::= &?

<operadorCaracter> ::= #?

Operaciones lógicas: ta yú pucá emé

En código:

```
##Definimos el booleano
afá b1 nicó tócu;
afá b2 nicó maíca;

## and
b1 ta b2;
→ maíca

## or
b1 yú b2;
→ tócu

## xor
b1 pucá b2;
→ tócu

## not
b1 emé
→ maíca
```

<operacionesLogicas> ::= id <operadorBooleano> id ;

<operadorBooleano> ::= ta

<operadorBooleano> ::= yú

<operadorBooleano> ::= pucá

<operadorBooleano> ::= emé

Operaciones de strings: Concatenar: catáqui / Length: lharátesufá / cortar: joné / recortar: nhalh / encontrar: cochéqui

En código:

```
##Definir las variables
ilo string1 nicó "Esto es un string";
ilo string2 nicó "el veloz murciélago";

##concatenar
string1 catáqui string2
→ "Esto es un stringel veloz murciélago"

## lenght
lharátesufá string1
→ 17

## cortar
joné[0:5] string1
→ "Esto e"

##recortar
nhalh[0:5] string2
→ "oz murciélago"

##encontrar
cochéqui["un"] string1
→ 8
```

<operacionesStrings> ::= id catáqui id

<operacionesStrings> ::= lharátesufá id

<operacionesStrings> ::= <operadorManejoString> [<literalEntera> : <literalEntera>] id

<operadorManejoString> ::= joné

<operadorManejoString> ::= nhalh

<S> ::= cochéqui [<literalString>] id

Operaciones con conjuntos: Agregar: erronh / Borrar:quené / Union: fuinhye / Interseccion: sulí / PerteneceA: pcalúfa / Vacío?: culíte

En código:

```
## agregar
erronh con1 'x5';
→ {'a1','x1','x2','x3','x4','x5'}

## borrar
quené con1 'x1';
→ {'a1','x2','x3','x4'}

## Union
fuinhye con1 con2
→ {'a1','x1','x2','x3','x4','a1','y1','y2','y3','y4'}

## intersección
sulí con1 con2
→ {'a1'}

## pertenece
pcalúfa con1 'a1'
→ tócu

## vacio
culíte con1
→ maíca
```

<operacionesConjuntos> ::= <operadorConjunto> id <literalX>

<operadorConjunto> ::= erronh

<operadorConjunto> ::= quemé

<operadorConjunto> ::= pcalúfa

<operacionesConjuntos> ::= <operadoConjunto> id id

<operadorConjunto> ::= sulí

<operadorConjunto> ::= fuinhye

<operacionesConjuntos> ::= <operadorConjunto> id

<operadorConjunto> ::= culíte>

Operaciones con archivos: Abrir: quijí / Cerrar: eloc / Escribir: lhioc / Leer: lhaic / Crear: quijérri / Asociar: láte

En código:

```
lhalámi cheto nicó {{ "c:\akw\tarea1.txt" , L }}

## abrir
quijí cheto;

##cerrar
eloc cheto;

## escribir
lhioc cheto "texto a escribir";

## crear
quijérri muffin;

## asociar
láte muffin cheto;
```

<manejoArchivo> ::= <operadorArchivo> id

<manejoArchivo> ::= <operadorArchivo> id id

<manejoArchivo> ::= <operadorArchivo> id <literalString>

<operadorArchivo> ::= quijí <operadorArchivo> ::= eloc

<operadorArchivo> ::= lhioc

<operadorArchivo> ::= quijérri

<operadorArchivo> ::= láte

Operaciones con números flotantes:|++ ++| |+| +|+

En código:

```
jané f1 nicó 1.23;
jané f2 nicó 41.77;

## suma
f1 |++ f|
→ 43.000

## resta
f1 ++| f2
→ -40.54

## multiplicacion
f1 |+| f2
→ 51.3771

## division
f1 +|+ f2
→ 0.00294469
```

<operacionesFlotantes> ::= id <operadorFlotante> id

<operacionesFlotantes> ::= <literalFlotante> <operadorFlotante> id

<operacionesFlotantes> ::= id <operadorFlotante> <literalFlotante>

<operadorFlotante> ::= |++

<operadorFlotante> ::= ++|

<operadorFlotante> ::= |+|

<operadorFlotante> ::= +|+

Operaciones de comparación: < > = >= <= ><

En código:

<operaracionesComparacion> ::= id <operadorComparacion> id

<operaracionesComparacion> ::= id <operadorComparacion> <literalX>

<operaracionesComparacion> ::= <literalX> <operadorComparacion> id

<operadorComparacion> ::= <

<operadorComparacion> ::= >

<operadorComparacion> ::= =

<operadorComparacion> ::= >=

<operadorComparacion> ::=<=

<operadorComparacion> ::= ><

Manejo de bloques de más de una instrucción: cojcoj … fuchí

En código:

```
cojcoj
    pítechátené i;
    íca i;
fuchí
```

<bloqueInstrucciones> ::= cojcoj <loop-instrucciones> fuchí

<loop-instrucciones> ::= ε

<loop-instrucciones> ::= <instruccion> <loop-instrucciones>

<instruccion> ::= <asigConst> | <asigTipo> | <asigVar> | <booleanoTipo> | <reg> | <llamada-rutina> | halt | <while> | <for> | <repeat-until> | <if-then-else>

Instrucción while

En código:

```
##ejemplo de un while
ótacá contador >= 0 sóta ## while(contador >= 5) do
cojcoj
    contador jené;       ## esto es un contador--
    aumentador taqué;    ## esto es un aumentador++
    taráne               ## continue (no es obligatorio)
fuchí
```

<bloque-while> ::= ótacá <expresión-booleana> sóta <loop-instrucciones>

Instrucción if

En código: similar a if-then-else pero sin el segundo bloque. En la siguiente se puede ver un ejemplo.

<bloque-if> ::= arú <expresión-booleana> nocófa <bloque-código>

Instrucción if-then-else

En código:

```
##ejemplo de if-then-else
arú aumentador > 5 nocófa    ## if (aumentador > 5) then
cojcoj
    aumentador := 0;
    contador := 0;
fuchí
tiáfa                        ## else
cojcoj
    aumentador := 5;
    contador := 5;
fuchí
```

<bloque-if-else> ::= arú <expresión-booleana> nocófa <bloque-código> tiáfa <bloque-código>

Instrucción switch

En código:

```
torré Letra      ## torré es switch
cojcoj
    ári 'a' :    ## ári es el case
    ári 'e' :
    ári 'i' :
    ári 'o' :
    ári 'u' :
        jalac tócu; ## jalac es return, o
        foróforóye; ## este es el break
    mailhíco:       ## este es el default
        jalac maíca;## return false
fuchí
```

<bloque-switch> ::= torré <identificador> cojcoj <loop-casos> fuchí

<loop-casos> ::= ári <literal> : <loop-instrucciones> <loop-casos>

<loop-casos> ::= ári <literal> : <loop-casos>

<loop-casos> ::= mailhíco : <loop-instrucciones>

Instrucción repeat-until

En código:

```
## Ejemplo de repeat-until
paípaí  ## do
    aumentador += 1;
yája aumentador < 10; ## while
```

<bloque-repeat-until> ::= paípaí <loop-instrucciones> yája <expresión-booleana>

Instrucción for

En código:

```
##ejemplo de un for
ajá chátené i := 5 Iha i >= 0 telele i jené sóta ## fo
cojcoj              ## el cojcoj al igual que el {} en
    pítechátené i;  ## píte es salida estandar o sea e
    íca i;          ## sizeof de i
fuchí
```

<for> ::= ajá <tiposDeDatos> <asiganciones> Iha <operaracionesComparacion> telele id

<DecIncremento> sóta <instruccion>

Instrucciones break, continue y halt

En código:

```
chátené contador nicó 10;
chátené aumentador nicó 0;

ótacá(contador >= 0) sóta
cojcoj
    contador jené;
    aumentador taqué;
    arú (aumentador = 5) nocófa
        foróforóye
    taráne
fuchí

lauc
```

<break> ::= foróforóye

<continue> ::= taráne

<halt> ::= lauc

Encabezado de funciones

<encabezadoFun> ::= macorróca <tipo> id ( )

Encabezado de procedimientos

<encabezadoProc> ::= óri id ( )

Manejo de parámetros formales

<parametroFormal> ::= (<tiposDeDatos> id, <tiposDeDatos> id, sui <tiposDeDatos> id)

Manejo de parámetros reales

<parámetrosReales> ::= id (a, b, c)

Instrucción return

<return> ::= jalac id;

<return> ::= jalac <literal> ;

<return> ::= jalac ;

Operación sizeof

<sizeof> ::= icá id;

Sistema de coerción de tipos

<coerción> ::= tirríque (<tiposDeDatos>) id ;

Salida Estándar

<salidaES> ::= lac<tiposDeDatos> id

Entrada Estándar

<entradaES> ::= píte<tiposDeDatos> id

Pruebas de análisis léxico

Test de comentarios

Nombre del archivo: test_comentarios.taf

Objetivo: probar la correcta detección de comentarios en las distintas partes del programa

```
--- exec:3.1.0:exec (default-cli) @ Lexico ---



Tipo: EOF  Lexema:   Linea:5  Columna:1

Compilacion correcta.
```

Este archivo solo genera el token de End Of File, ya que los comentarios son ignorados por el Scanner

Test de Seccion de constantes

Nombre del archivo: test_seccion_constantes.taf

Objetivo: probar la correcta detección de comentarios en las distintas partes del programa

```
Tipo: SECCION  Lexema: octar□  Linea:2  Columna:1
Tipo: ASIGNACION  Lexema: arr□  Linea:4  Columna:2
Tipo: IDENTIFICADOR  Lexema: x1  Linea:4  Columna:7
Tipo: LITERAL_ENTERO  Lexema: 45  Linea:4  Columna:10
Tipo: TERMINADOR  Lexema: ;  Linea:4  Columna:12
Tipo: ASIGNACION  Lexema: arr□  Linea:5  Columna:2
Tipo: IDENTIFICADOR  Lexema: x2  Linea:5  Columna:7
Tipo: LITERAL_FLOTANTE  Lexema: 1.5  Linea:5  Columna:10
```

genera los tokens correctamente (el símbolo □ es porque la consola de java no soporta tildes)

Muro de ladrillos del ejemplo:

Tokens

Lexema	Fila	Columna	Tipo
octará	2	1	SECCION
arrá	4	2	ASIGNACION
x1	4	7	IDENTIFICADOR
45	4	10	LITERAL_ENTERO
;	4	12	TERMINADOR
arrá	5	2	ASIGNACION
x2	5	7	IDENTIFICADOR
1.5	5	10	LITERAL_FLOTANTE
;	5	13	ERROR
	7	1	EOF

Contador de Tokens

- TERMINADOR: 1
- LITERAL_FLOTANTE: 1
- LITERAL_ENTERO: 1
- SECCION: 1
- ERROR: 1
- ASIGNACION: 2
- IDENTIFICADOR: 2
- EOF: 1

Pruebas de análisis sintáctico

Se crearon una serie de archivos .taf con el fin de probar el análisis de la estructura sintáctica de los programas en Táfac. En esta sección se detallará el contenido de cada uno y su intención

Test de comentarios

Nombre del archivo: test_comentarios.taf

Objetivo: probar la correcta detección de comentarios en las distintas partes del programa

Test de fors

Nombre del archivo: test_for.taf

Objetivo: probar tanto la correcta detección de estructuras de ciclos for en el programa, así como sus posibles errores

Test de ifs

Nombre del archivo: test_ifs.taf

Objetivo: probar la correcta detección de estructuras de if then else dentro del programa, también sus posibles errores.

Test de inicio incorrecto

Nombre del archivo: test_mal_inicio.taf

Objetivo: probar la correcta iniciación del programa. Si el programa no tiene el iniciador seguido explícitamente de un nombre, da error.

Test de repeat until

Nombre del archivo: test_repeat_until.taf

Objetivo: probar la correcta detección de la estructura repeat until y sus posibles errores como falta de expresión booleana.

Test de sección de constantes

Nombre del archivo: test_sin_constantes.taf

Objetivo:Al ser la sección de constantes opcional, esta prueba tiene el fin de probar el correcto funcionamiento en el caso de que esta sección no esté definida, es decir, que el programa no encuentre errores sintácticos.

Test de falta de inicio

Nombre del archivo: test_sin_inicio.taf

Objetivo: probar la correcta detección de la falta de iniciación del programa, es decir, no existe la cláusula que inicia el programa con su nombre.

Test de sección de prototipos

Nombre del archivo: test_sin_prototipos.taf

Objetivo: Al ser la sección de prototipos opcional, esta prueba tiene el fin de probar el correcto funcionamiento del caso de que esta sección no esté definida, es decir, que el programa no encuetre errores sintácticos.

Test de sección de rutinas

Nombre del archivo: test_sin_rutinas.taf

Objetivo: Al ser la sección de rutinas opcional, esta prueba tiene el fin de probar el funcionamiento del caso de que esta sección no esté definida, es decir, que el programa no encuentre errores sintácticos.

Test de sección de tipos

Nombre del archivo: test_sin_tipos.taf

Objetivo: Al ser la sección de tipos opcional, esta prueba tiene el fin de probar el funcionamiento del caso de que esta sección no esté definida, es decir, que el programa no encuentre errores sintácticos.

Test de sección de variables

Nombre del archivo:test_sin_variables.taf

Objetivo: Al ser la sección de variables opcional, esta prueba tiene el fin de probar el funcionamiento del caso de que esta sección no esté definida, es decir, que el programa no encuentre errores sintácticos.

Test de while

Nombre del archivo: test_while.taf

Objetivo: Probar la correcta detección de la estructura while a lo largo del programa, así como sus posibles errores sintácticos.

Funcionamiento del scanner

El Scanner es una clase de Java con distintos métodos para conseguir un análisis de cada token del código a compilar. Los distintos métodos poseen manejo de errores mediante las excepciones proveídas por Java.

El método principal de este componente es llamado scan(). Esta función crea una lista de tokens y la va llenando con los tokens que saca del archivo de texto mediante el método demeToken() en ciclo do-while, con condición de parada el token EOF (end of file), ya que este siempre está y siempre es el último.

El funcionamiento del corazón del Scaner está en el método demeToken(), este construye el lexema del token con los caracteres extraidos del archivo .taf, para hacer esto tiene un método llamado demeCaracter(), que va concatenando mientras no haya un salto de línea, un espacio, o una tabulación (ver método esEspacio() y saltaEspacios()). Una vez construido el lexema sigue la parte de clasificiar el lexema en un tipo de token, para eso hay un gran Switch con todos los elementos y palabras reservadas del lenguaje, cada uno retorna un tipo en específico. Así el token se construye con un tipo, un lexema, una fila y una columna

```
switch (lexemaStr)
{
    //Secciones
    case "nhequéquequé","octará","aíca","tilhtilh","chá","purú" ->
    {
        return new Token(TipoToken.SECCION, lexemaStr, line, column);
    }
    //Tipos de dato
    case "chátené", "ílo", "afá","apúchá","lhalámi","jané","coyé","poréteca" ->
    {
        return new Token(TipoToken.TIPO_DATO, lexemaStr, line, column);
    }
```

A lo largo de este método se usan otras funciones como obtenerLineaError(), demeCaracter(), reportarError(), obtenerLineaError() que poseen funcionamientos más sencillos y cuyo nombre es completamente autodescriptivo. Estos errores están ligados únicamente a los errores que se pueden encontrar en esta etapa del compilador, como el de caracter monstruo, que es cuando hay un caracter que no está definido en los tipos de tokens, puede probar esto añadiendo un ”¡” al archivo .taf

```
default ->
{
    String lineaError = obtenerLineaError(line);
    reportarError("\nError, caracter monstruo", lineaError, column);
    return new Token(TipoToken.ERROR, lexemaStr, line, column);
}
```

Funcionamiento del parser

El Parser es una clase de Java con disintos métodos para conseguir un análisis sintáctico de la lista de tokens obtenida del paso anterior.

El funcionamiento del Parser es más complejo que el Scanner, para entender esto primero se deben explicar todas las clases que lo conforman

Grammar: esta es la clase encargada de crear la gramática del lenguaje diseñada en la etapa 0, esta almacenará una lista de no terminales, terminales y producciones, además del símbolo inicial de cada producción, el <S>. Su método principal es getProductionsForNonTerminal(String nonTerminal), el cual retornará una lista con todas las producciones que empiecen con el no terminal dado en el parámetro

```
public List<Producción> getProductionsForNonTerminal(String nonTerminal) {
    List<Producción> result = new ArrayList<>();
    for (Producción production : productions) {
        if (production.getNoTerminal().equals(nonTerminal)) {
            result.add(production);
        }
    }
    return result;
}
```

FirstFollow: esta clase crea el conjunto de first y follow, para esto tiene tres variables principales, Grammar que es una Objeto de tipo Grammar, firstSets que es un Map igual que followSets. El corazón de esta clase está en los calculate, una para follow y otra para First

```java
public Set<String> first(List<String> sequence) {
    Set<String> result = new HashSet<>();
    if (sequence.isEmpty()) {
        result.add("ε");
        return result;
    }

    String firstSymbol = sequence.get(0);
    if (grammar.getTerminals().contains(firstSymbol)) {
        result.add(firstSymbol);
    } else {
        result.addAll(firstSets.get(firstSymbol));
        for (int i = 1; i < sequence.size() && firstSets.get(sequence.get(i - 1)).contains("ε"); i++) {
            result.addAll(firstSets.get(sequence.get(i)));
            result.remove("ε");
        }
        if (sequence.stream().allMatch(symbol -> firstSets.get(symbol).contains("ε"))) {
            result.add("ε");
        }
    }
    return result;
}
```

result es la variable que devuelve el conjunto de first calculado, para ello revisa la gramática en busca de terminales que comiencen con el primer símbolo de la secuencia dada. También hace manejo del epsilon.

```java
public void calculateFollow() {
    followSets.get(grammar.getStartSymbol()).add("$");
    boolean changed;
    do {
        changed = false;
        for (Producción production : grammar.getProductions()) {
            String nonTerminal = production.getNoTerminal();
            List<String> sequence = production.getSecuencia();

            for (int i = 0; i < sequence.size(); i++) {
                String symbol = sequence.get(i);
                if (grammar.getNonTerminals().contains(symbol)) {
                    Set<String> followSet = followSets.get(symbol);
                    int initialSize = followSet.size();

                    if (i + 1 < sequence.size()) {
                        List<String> restSequence = sequence.subList(i + 1, sequence.size());
                        Set<String> firstOfRest = first(restSequence);
                        followSet.addAll(firstOfRest);
                        followSet.remove("ε");
                        if (firstOfRest.contains("ε")) {
                            followSet.addAll(followSets.get(nonTerminal));
                        }
                    } else {
                        followSet.addAll(followSets.get(nonTerminal));
                    }

                    if (followSet.size() > initialSize) {
                        changed = true;
                    }
                }
            }
        }
    } while (changed);
}
```

El follow no necesita tomar parámetros, además que usa el $ como símbolo inicial, este itera sobre la secuencia de las producciones y genera la tabla de folllow

PredictCalculator: esta clase es exclusiva para calcular el predict, posee tres variables principales, una de tipo Grammar, una de tipo FirstFollow y un Map que tendrá los sets de predicts

```
public PredictCalculator(Grammar grammar, FirstFollow firstFollowCalculator) {
    this.grammar = grammar;
    this.firstFollowCalculator = firstFollowCalculator;
    this.predictSets = new HashMap<>();

    for (Producción production : grammar.getProductions()) {
        predictSets.put(production, new HashSet<>());
    }
}

public void calculatePredict() {
    for (Producción production : grammar.getProductions()) {
        List<String> sequence = production.getSecuencia();
        Set<String> firstSet = firstFollowCalculator.first(sequence);

        if (firstSet.contains("ε")) {
            firstSet.remove("ε");
            firstSet.addAll(firstFollowCalculator.getFollowSets().get(production.getNoTerminal()));
        }

        predictSets.get(production).addAll(firstSet);
    }
}
```

Su corazón está en éstos dos métodos, PredictCalculator carga las variables principales de la clase mientras que carga de producciones el conjunto de predicts y calculatePredict toma las producciones añadidas, remueve los epsilon y añade al conjunto de first las no terminales de las producciones del conjunto de follow de la clase que los calcula.

Producción: esta es una clase simple que guarda una produccion con su secuencia

```
public class Producción {
    private String noTerminal;
    private List<String> secuencia;

    public Producción(String noTerminal, List<String> secuencia) {
        this.noTerminal = noTerminal;
        this.secuencia = secuencia;
    }

    public String getNoTerminal() {
        return noTerminal;
    }

    public List<String> getSecuencia() {
        return secuencia;
    }
```

Posee métodos para obtener el no terminal inicial y la secuencia de este.

ParsingTable: esta clase es la que construye la tabla de parsing, aunque no es quien la llena

```java
public class ParsingTable {
    private Map<String, Map<String, Integer>> table;

    public ParsingTable() {
        table = new HashMap<>();
    }

    public void addEntry(String nonTerminal, String terminal, int productionNumber) {
        table.computeIfAbsent(nonTerminal, k -> new HashMap<>()).put(terminal, productionNumber);
    }

    public int getEntry(String nonTerminal, String terminal) {
        return table.getOrDefault(nonTerminal, new HashMap<>()).getOrDefault(terminal, -1);
    }

    public void printTable() {
        for (Map.Entry<String, Map<String, Integer>> row : table.entrySet()) {
            System.out.print(row.getKey() + ": ");
            for (Map.Entry<String, Integer> col : row.getValue().entrySet()) {
                System.out.print(col.getKey() + "->" + col.getValue() + " ");
            }
            System.out.println();
        }
    }
}
```

Esta clase es más para definir su estructura y los métodos que permite para su control

ParsingTableBuilder: este es el corazon de la tabla de parsing, es quien se encarga de construirla a partir del conjunto de predicts

```java
public class ParsingTableBuilder {
    private Grammar grammar;
    private PredictCalculator predictCalculator;
    private ParsingTable parsingTable;

    public ParsingTableBuilder(Grammar grammar, PredictCalculator predictCalculator) {
        this.grammar = grammar;
        this.predictCalculator = predictCalculator;
        this.parsingTable = new ParsingTable();

        buildTable();
    }

    private void buildTable() {
        Map<Producción, Set<String>> predictSets = predictCalculator.getPredictSets();
        for (Map.Entry<Producción, Set<String>> entry : predictSets.entrySet()) {
            Producción production = entry.getKey();
            Set<String> predictSet = entry.getValue();
            for (String terminal : predictSet) {
                parsingTable.addEntry(production.getNoTerminal(), terminal, grammar.getProductions().indexOf(production));
            }
        }
    }

    public ParsingTable getParsingTable() {
        return parsingTable;
    }
}
```

Parser2: este es el corazon del parser, se llama 2 porque es la versión 2 de este, la vieja aún se puede encontrar en la clase Parser y es de quien está basado todas las demás clases. Esta clase posee 6 variables principales

```
private List<Token> tokens;
private List<Producción> producciones;
private ParsingTable parsingTable;
private Stack<String> pila;
private int currentIndex;
private Grammar grammar;
```

La lista de tokens a parsear, las producciones de la gramática, la tabla de parsing, una pila de control, un index para manejar el total de tokens y la gramática. El corazón del corazón es el método parse(), este se encarga de tomar token a token y compararlo con la pila de parsing, quien contiene las reglas gramaticales, el controlador principal de este método es saber si el dato actual en la pila es un terminal o un no terminal, si es un terminal lo compara directamente con el token tomado, ya sea por su tipo o lexema, si no deberá recorrer la secuencia en la gramática hasta encontrar una producción que calce con el token tomado. Si llegó al token EOF sin errores, reporta que todo está en orden, en caso de que el orden de tokens no esté definido, reportará el error diciendo que se encontró y que se esperaba.

ParserMain: como su nombre lo indica, este es el cerebro del parser. Aquí se hace uso de todas las clases anteriores para parsear el código

```
Grammar grammar = new Grammar();
grammar.setStartSymbol("<S>");

grammar.addNonTerminal("<S>");
grammar.addNonTerminal("<loop-constantes>");
grammar.addNonTerminal("<literal>");
grammar.addTerminal("octará");
grammar.addTerminal("arrá");
grammar.addTerminal("IDENTIFICADOR");
grammar.addTerminal("LITERAL_FLOTANTE");
grammar.addTerminal("LITERAL_BOOLEANA");
grammar.addTerminal("LITERAL_ENTERO");
grammar.addTerminal("COMA");
grammar.addTerminal("TERMINADOR");
grammar.addTerminal("TIPO_DATO");

grammar.addNonTerminal("<loop-definicion>");
grammar.addTerminal("aíca");
grammar.addTerminal("putú");

grammar.addTerminal("tilhtilh");
grammar.addTerminal("nicó");
grammar.addNonTerminal("<seccion-variables>");

grammar.addTerminal("nhequéquequé");
```

primero se deben añadir manualmente los terminales y no terminales, como nomenclatura, los no terminales estará entre <>, los terminales estarán en mayúscula si se espera leer el tipo de token y en minúscula si se espera leer el lexema del token. También se añade el símbolo inicial

```
//nombre del programa
grammar.addProduction("<S>", Arrays.asList("nhequéquequé","IDENTIFICADOR","TERMINADOR"));
// Sección de constantes
grammar.addProduction("<S>", Arrays.asList("octará","arrá","<loop-constantes>"));
//grammar.addProduction("<loop-constantes>", Arrays.asList("IDENTIFICADOR", "<literal>", "COMA", "<loop-constantes>"));
grammar.addProduction("<loop-constantes>", Arrays.asList("IDENTIFICADOR", "<literal>", "TERMINADOR"));
grammar.addProduction("<literal>", Arrays.asList("LITERAL_FLOTANTE"));
grammar.addProduction("<literal>", Arrays.asList("LITERAL_BOOLEANA"));
grammar.addProduction("<literal>", Arrays.asList("LITERAL_ENTERO"));

//sección de tipos
grammar.addProduction("<S>", Arrays.asList("aíca","<loop-definicion>"));
grammar.addProduction("<loop-definicion>", Arrays.asList("putú","IDENTIFICADOR","TIPO_DATO","TERMINADOR"));

//sección de variables
grammar.addProduction("<S>", Arrays.asList("tilhtilh","<seccion-variables>"));
grammar.addProduction("<seccion-variables>", Arrays.asList("TIPO_DATO","IDENTIFICADOR","nicó","<literal>","TERMINADOR"));

//asiganaciones
grammar.addProduction("<S>", Arrays.asList("IDENTIFICADOR","ASIGNACION","<literal>"));

//operador entero
grammar.addProduction("<S>", Arrays.asList("IDENTIFICADOR","OPERADOR_ENTERO","IDENTIFICADOR","TERMINADOR"));
```

Luego se añade la gramática, cada familia de gramática iniciará con <S>, seguido de un Array que contiene el resto de elementos de la gramática

```
// Calcular First y Follow
FirstFollow firstFollowCalculator = new FirstFollow(grammar);
firstFollowCalculator.calculateFirst();
firstFollowCalculator.calculateFollow();

// Calcular Predict
PredictCalculator predictCalculator = new PredictCalculator(grammar, firstFollowCalculator);
predictCalculator.calculatePredict();

// Construir la tabla de parsing
ParsingTableBuilder parsingTableBuilder = new ParsingTableBuilder(grammar, predictCalculator);
ParsingTable parsingTable = parsingTableBuilder.getParsingTable();

//obtiene los tokens del Scanner
List<Token> tokensTrue = Scanner.scan();

// Crear el parser y ejecutar el análisis sintáctico
Parser2 parser = new Parser2(tokensTrue, grammar.getProductions(), parsingTable, grammar);
parser.parse();
```

Por último, se ejecuta en orden cada clase del parser para al final crear el objeto parser que hará uso del resto de objetos de la otra clase, el paso final es ejecutar el corazón final.

En esta clase también está la llamada a la tabla de símbolos del analizador sintáctico

```
SymbolTable tabla = new SymbolTable();

try {
    tabla.fillTable(tokensTrue);
} catch (SymbolTableCollisionException e) {
    System.out.println(x: e.toString());
}


System.out.println(x: "----------------Tabla de simbolos----------------");
tabla.printTable();
//System.out.println(tokensTrue.size());
```

simplemente se llena con los tokens obtenidos previamente del Scanner y se ejecuta el método que imprime la tabla de símbolos, con manejo de errores en caso de colisión, debido a que está montada sobre una tabla hash

Tabla de símbolos

La tabla de símbolos es una clase de Java con un HashTable cuya llave es un objeto compuesto del Scope y el Identificador y el valor es un objeto compuesto de los datos del símbolo obtenido.

Clase ScopeIdentifier: esta clase contienen la información utilizada para crear la llave de la tabla de símbolos, es decir, el identificador del símbolo en cuestión y el alcance de dicho símbolo (si es global o a qué bloque de código pertenece) por lo que una variable, constante, etc con el mismo nombre y alcance serán ignorados por el compilador. Además, posee los métodos constructores, setter y getter, además de la sobreescritura de los métodos por defecto para generar el hash y para comprar un objeto con este, esto con el fin de evitar agregar a la tabla de símbolos, variables con el mismo nombre y alcance.

```
public class ScopeIdentifier {
    private String scope;
    private String identifier;
```

Clase SymbolInfo: esta clase contiene la información relevante a la declaración realizada de una función, variable, constante, etc. Guarda el tipo de dato, información adicional (que usualmente es el valor guardado en la variable), la línea donde se declaró y si es o no constante para saber si se puede o no modificar. Posee sus métodos constructores, setter y getter. Esta clase será utilizada como valor en la tabla de símbolos dada una llave.

```
public class SymbolInfo {
    String dataType;
    String additionalInfo;
    int line;
    boolean isConstant;
```

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    ScopeIdentifier that = (ScopeIdentifier) o;
    return scope.equals(anObject: that.scope) && identifier.equals(anObject: that.identifier);
}

@Override
public int hashCode() {
    return Objects.hash(values:scope, values:identifier);
}
```

Clase SymbolTable: en esta clase es donde está ubicada la tabla de símbolos y la función para llenar a esta. El único atributo de esta clase es globalSymbolTable, que es un HashTable de la librería de utilidades de java. La función hash de este fue modificada para

que no permita agregar items con la misma llave. El método fillTable recibe la lista de tokens después de análisis sintáctico y léxico, por lo que se supone ya están correctos. Usa unos booleanos para determinar en qué sección del código fuente se encuentra y de esta manera saber cómo agregar los ítems a la tabla. Posteriormente, al encontrar indicativo de declaración, utiliza los tokens ordenados por el parser para encontrar el identificador de las variables, su tipo de dato y su valor si lo posee, de lo contrario utiliza un default. Finalmente, retorna el objeto globalSymbolTable para su posterior utilización.

Por otra parte, posee dos métodos para detectar si un lexema es un entero o un flotante, esto con el fin de encontrar el tipo de dato de la declaración en la sección de variables, puesto que no viene explícito en el fuente.

Esta es la sencilla declaración de la clase:

```
public class SymbolTable {
    Hashtable<ScopeIdentifier, SymbolInfo> globalSymbolTable;
```

Estos son los métodos para saber el tipo de dato de la sección de variables:

```
public static boolean isInteger(String value) {
    try {
        Integer.parseInt(s: value);
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
}

public static boolean isFloat(String value) {
try {
    Float.parseFloat(s: value);
    return true;
} catch (NumberFormatException e) {
    return false;
}
```

Esta es la función de llenado de la tabla:

```java
public Hashtable<ScopeIdentifier, SymbolInfo> fillTable(List<Token> tokensTrue) throws SymbolTableCollisionException{
    boolean programName = false;
    boolean constSeccion = false;
    boolean typesSeccion = false;
    boolean variablesSeccion = false;
    boolean prototypesSeccion = false;
    boolean rutinesSeccion = false;
    String programNameStr = "";
    int tokensAmount = tokensTrue.size();

    for (int i = 0; i < tokensAmount; i++) {
        Token tmpToken = tokensTrue.get(index: i);
        if (tmpToken.getTipo() == TipoToken.SECCION) {
            if (!programName) { //definicion de nombre del programa
                programName = true;
                i = i + 2;
            } else if (tmpToken.getLexema().equals(anObject: "octará")) { //seccion de constantes
                constSeccion = true;
                typesSeccion = false;
                variablesSeccion = false;
                prototypesSeccion = false;
                rutinesSeccion = false;
            } else if (tmpToken.getLexema().equals(anObject: "aíca")) { //seccion de tipos
                constSeccion = false;
                typesSeccion = true;
                variablesSeccion = false;
                prototypesSeccion = false;
                rutinesSeccion = false;
            } else if (tmpToken.getLexema().equals(anObject: "tilhtilh")) { //seccion de variables
                constSeccion = false;
                typesSeccion = false;
                variablesSeccion = true;
```

```java
                rutinesSeccion = false;
            } else if (tmpToken.getLexema().equals(anObject: "chá")) { //seccion de prototipos (.h) en C
                constSeccion = false;
                typesSeccion = false;
                variablesSeccion = false;
                prototypesSeccion = true;
                rutinesSeccion = false;
            } else if (tmpToken.getLexema().equals(anObject: "purú")) { //seccion de rutinas
                constSeccion = false;
                typesSeccion = false;
                variablesSeccion = false;
                prototypesSeccion = false;
                rutinesSeccion = true;
            }
        } else if (constSeccion && tmpToken.getTipo() == TipoToken.ASIGNACION) {
            Token constNameToken = tokensTrue.get(i+1);
            Token constValueToken = tokensTrue.get(i+2);
            String value = constValueToken.getLexema();
            String dataType;
            if (value.startsWith(prefix:"\"")) {
                dataType = "ílo";
            } else if (value.startsWith(prefix:"\'")) {
                dataType = "malhióca";
            } else if (value.equals(anObject: "tócu") || value.equals(anObject: "maíca")) {
                dataType = "afá";
            } else if (isInteger(value)) {
                dataType = "chátené";
            } else if (isFloat(value)) {
                dataType = "jané";
            } else {
                dataType = "undefined";
            }
            ScopeIdentifier tmpIdentifier = new ScopeIdentifier(scope: "Global", identifier: constNameToken.getLexema());
```

```
                ;
                ScopeIdentifier tmpIdentifier = new ScopeIdentifier(scope: "Global",identifier: constNameToken.getLexema());
                SymbolInfo tmpInfo = new SymbolInfo(dataType,additionalInfo: value,line: constValueToken.getLinea(), isConstant: true);
                this.addSymbol(key: tmpIdentifier, info: tmpInfo);
            } else if (typesSeccion && tmpToken.getTipo() == TipoToken.ASIGNACION) {
                Token constNameToken = tokensTrue.get(i+1);
                Token constValueToken = tokensTrue.get(i+2);
                String value = constValueToken.getLexema();
                ScopeIdentifier tmpIdentifier = new ScopeIdentifier(scope: "Global",identifier: constNameToken.getLexema());
                SymbolInfo tmpInfo = new SymbolInfo(dataType: value,additionalInfo: "",line: constValueToken.getLinea(), isConstant: true);
                this.addSymbol(key: tmpIdentifier, info: tmpInfo);
            } else if (variablesSeccion && tmpToken.getTipo() == TipoToken.TIPO_DATO) {
                String dataType = tmpToken.getLexema();
                String identifier = tokensTrue.get(i+1).getLexema();
                String value = tokensTrue.get(i+3).getLexema();
                ScopeIdentifier tmpIdentifier = new ScopeIdentifier(scope: "Global",identifier);
                SymbolInfo tmpInfo = new SymbolInfo(dataType,additionalInfo: value,line: tmpToken.getLinea(),isConstant: false);
                this.addSymbol(key: tmpIdentifier, info: tmpInfo);
            }
        }
        return this.globalSymbolTable;
    }
```