UNIVERSIDAD DE CANTABRIA

Lenguajes de Programación

G1662

Compilador de C

Carlos Velázquez Fernández

June 3, 2023

1 Introducción

El objetivo de este proyecto es realizar un compilador funcional del lenguaje de programación C. Dada la completitud y complejidad del lenguaje, no se han implementado todos los aspectos de este. Sin embargo, sí se han llevado a cabo las funcionalidades clave que lo caracterizan, consiguiendo así un lenguaje básico pero con una gama de instrucciones suficiente para soportar cualquier programa.

Podemos dividir el compilador en cuatro partes: analizador léxico (lexer), analizador sintáctico (parser), analizador semántico e intérprete. Cada parte ha sido testada de manera individual y por medio de ficheros de prueba con el objetivo de asegurar de que todo funciona y devuelve el output esperado.

La mayor parte del código se encuentra en el anexo y se irá referenciando a medida que se mencione en el documento. No es necesaria su lectura para comprender las explicaciones, sin embargo, se ha añadido para facilitar su consulta si se desea comprobar algo en específico, sin necesidad de navegar entre distintos ficheros y partiendo de un contexto.

2 Analizador Léxico

El analizador léxico tokenizará los programas, separando y clasificando cada token en función de si se trata de una palabra clave, identificador de una variable, número...

Para llevar a cabo esta tarea, primero se deben definir las keywords de nuestro lenguaje. La selección en este compilador es la siguiente:

break	else	long
case	float	return
char	for	switch
continue	if	void
double	int	while

Como se puede observar, estas palabras consisten en las instrucciones básicas y los tipos de los datos (a excepción de algunas como *for*, *break* o *continue* que se han añadido por comodidad para el usuario). Una vez hecho esto, identificaremos los literales (o combinaciones de estos) de nuestro lenguaje:

(	+	++	<	;
)	-	–	>	,
{	/	+=	<=	:
}	*	-=	>=	!
.	%	=	==	

El próximo paso será recoger estos tokens por medio de expresiones regulares (6.1.1) atendiendo a una serie de normas:

- En C, todas las keywords se escriben completamente en minúsculas.
- Los nombres de las variables deberán empezar por un carácter numérico (mayúscula o minúscula) o un guión bajo, pudiendo continuar además de por los anteriores, por caracteres numéricos.

De forma auxiliar, se han implementado dos lexers más, uno para comentarios (6.1.2) y otro para strings (6.1.3).

Podemos distinguir dos tipos de **comentarios**: de una o de múltiples lineas. Los primeros comienzan por "//" y son fáciles de procesar, ya que solo tenemos que indicar al compilador que pase a la siguiente linea. Los comentarios multilinea, sin embargo, son más complejos ya que deben estar delimitados por un principio ("/*") y un fin ("*/"). El compilador debe ignorar todo lo que hay en medio.

En cuanto a los **strings**, debemos hacer un matiz. En C no existe el tipo de datos *string* como uno de los tipos básicos, pero sí contamos con *char*. Los strings se forman creando arrays de caracteres, y aunque en este compilador no vamos considerar la implementación de listas, guardar datos en

forma de string nos será útil en el futuro, por ejemplo, para imprimir texto por pantalla cuando definamos manualmente la función *printf*. Cada string empieza y termina por comillas dobles y almacena todo el contenido en forma de texto, por lo que no hay que procesarlo.

El **analizador léxico principal** (6.1.4) será el encargado de recorrer el programa realizando la clasificación de los tokens y llamando a los lexers secundarios cuando sea necesario. Finalmente, devolverá una lista con la linea en la que se encuentra cada token, su tipo (según nuestras expresiones regulares) y su valor (el token en sí mismo).

Como medida de seguridad para comprobar el correcto funcionamiento, además de los fichero de prueba, se han programado funciones de error (6.1.5 para múltiples casos que muestran al usuario cuál es el error y dónde se ubica.

3 Analizador Sintáctico

El analizador sintáctico, o parser, será el encargado de construir el árbol sintáctico del lenguaje. Siguiendo una serie de reglas, identificará los elementos del código, desde las constantes como números o nombres de variables, hasta los métodos completos o el propio programa en sí.

La especificación de la sintaxis del lenguaje implementada se ha realizado mediante notación de expresiones regulares. Los elementos entre corchetes simples ([ ]) se consideran opcionales y los elementos entre corchetes dobles ([[ ]]) se usan para agrupar expresiones. Los espacios entre elementos no son relevantes en la sintaxis del lenguaje, pero se han incluido en el árbol por claridad.

El árbol sintáctico construido para el compilador se puede ver en la figura 1. De igual forma, a continuación se explicará de forma resumida y más comprensible:

1. Partimos desde la raíz ***programa*** (6.2.1) que se compone de cero o más características.
2. Una *característica* (6.2.2) podrá ser un atributo o un método. Es decir, un programa se compone de funciones y variables declaradas y/o inicializadas fuera de estos (variables globales).
3. Llamamos ***atributo*** (6.2.3) a las variables que no están contenidas en ningún método.
4. Un ***método*** (6.2.4) constituye una función con su tipo, sus parámetros (también indicando el tipo) y su cuerpo. Podrá tomar cero o más argumentos y contener cero o más expresiones en su cuerpo. El método no tiene por qué estar inicializado, sino que puede definirse únicamente su prototipo.
5. Una ***declaración*** (6.2.5) consiste en el par tipo de dato e identificador. Los tipos de datos son un conjunto de keywords del lenguaje y el identificador es el nombre de la variable.
6. Las ***expresiones*** (6.2.6) son los elementos más completos ya que constituyen la mayoría de instrucciones. Estas recogen las expresiones aritméticas (+, -, *, /), relacionales (>, <, >=, <=, ==, !=), lógicas (!), condicionales (if, switch), bucles (while, for), asignaciones...
7. Finalmente, contamos con los ***tipos*** (6.2.7) que se basan en los principales tipos de datos que soporta el lenguaje (por simplicidad para el programa, todos los tipos decimales se convierten a “float”, ya que no implementaremos distintas precisiones para ellos).

Para acabar, debemos tener en cuenta la **precedencia**, es decir, la anteposición del cálculo o evaluación de determinados elementos frente a otros, atendiendo a cómo queremos que estos se comporten. El orden de estas operaciones lo determinan sus operandos y representado de mayor a menor será:

Precedencia
* /
+ -
< > <= >= == ! =
!
=

programa ::= [[*característica*]]*

característica ::= *atributo*

| *método*

atributo ::= *declaración* [= *expr*]

método ::= *declaración* ([*declaración* [[, *declaración*]]*]) { [[*expr*]]* }

| *declaración* ([*declaración* [[, *declaración*]]*]) ;

declaración ::= *tipo* IDENTIFIER

expr ::= *declaración* [= *expr*] ;

| IDENTIFIER = *expr* ;

| IDENTIFIER += *expr* [;]

| IDENTIFIER -= *expr* [;]

| IDENTIFIER ++ [;]

| IDENTIFIER - - [;]

| IDENTIFIER ([*expr* [[, *expr*]]*])

| if (*expr*) { [[*expr*]]* } [[else if (*expr*) { [[*expr*]]* }]]* [else { [[*expr*]]* }]

| while (*expr*) { [[*expr*]]* }

| for (*expr expr* ; *expr*) { [[*expr*]]* }

| switch (*expr*) { [[case *expr* : [[*expr*]]*]]+ }

| *expr* + *expr*

| *expr* − *expr*

| *expr* * *expr*

| *expr* / *expr*

| *expr* % *expr*

| *expr* == *expr*

| *expr* < *expr*

| *expr* <= *expr*

| *expr* > *expr*

| *expr* >= *expr*

| ! *expr*

| (*expr*)

| return *expr* ;

| break ;

| continue ;

| IDENTIFIER

tipo ::= INT

| CHAR

| VOID

| FLOAT

| DOUBLE

| LONG

Figure 1: Árbol sintáctico de C

4 Analizador Semántico

Aunque la sintaxis del programa sea correcta, necesitamos comprobar que su semántica también lo es. Para ello, recorreremos el árbol generado por el parser desde las hojas hasta la raíz comprobando que todos los tipos son correctos, las variables o métodos usados están declarados, existe un *main* en el programa, etc.

Cada nodo contará con su propia función *Tipo* que realizará estas comprobaciones. Además, contamos con estructuras como una pila en la que guardaremos los ámbitos generados (diccionario con las variables que declaramos y sus tipos) o un diccionario en el que guardaremos los métodos (junto a sus argumentos y tipos de retorno).

- La clase **Ambito** (6.3.1) contendrá las estructuras de datos que usaremos desde el analizador, así como todos los métodos necesarios para añadir o sacar ámbitos de la pila, añadir variables o acceder a ellas.
- La clase **Programa** (6.3.2) realizará la mayor parte del trabajo, encargándose de múltiples comparaciones. Primero crea un objeto ámbito y recorre el programa una vez añadiendo los atributos y los métodos a sus estructuras correspondientes. Mientras se lleva a cabo este proceso, también se comprueba que no haya métodos o atributos redefinidos, argumentos repetidos en los métodos y la existencia de un método *main*. Una vez realizado esto, lo vuelve a recorrer pero ahora llamando a la función *Tipo* de cada característica.
- La clase **Atributo** (6.3.3) simplemente comprueba que, en caso de haber una asignación, el tipo de las expresiones coincide con el de la variable.
- La clase **Metodo** (6.3.4) crea un nuevo ámbito al que añade todos sus parámetros. Comprueba que el tipo de la expresión que retorna coincide con el indicado en la definición y llama al *Tipo* de las expresiones de su cuerpo.
- Las **expresiones aritméticas** (6.3.5) comprueban que ambos operandos son del mismo tipo y este además es numérico.
- Las **expresiones relacionales y lógicas** (6.3.6) (cuyo tipo propio es "bool") comprueban que ambas partes tienen el mismo tipo y este además es numérico (excepto == y !=).
- Las **sentencias if, switch, while** y **for** (6.3.7) (todas de tipo void) crean nuevos ámbitos y comprueban que el tipo de sus condiciones es bool.
- La clase **Asignacion** (6.3.8), al igual que Atributo, comprueba que el tipo de la expresión asignada coincide con el de la declaración.
- La clase **LlamadaMetodo** (6.3.9) comprueba si el método está definido y que los tipos de las expresiones de los argumentos coinciden con los que se espera que tome dicho método.
- La clase **Objeto** (6.3.10) comprueba que la variable está definida en el ámbito actual o en los superiores.
- Las clases de los **tipos básicos** (6.3.11) simplemente asignan su propio tipo.
- La clases **Retorno** (6.3.12), **Break** y **Continue** tienen tipo void y no hacen comprobaciones.

5 Intérprete

Nuestro programa ya es correcto tanto sintáctica como semánticamente, por lo que es hora de probarlo. El próximo paso será indicarle al computador de alguna manera qué debe hacer al encontrarse con las distintas instrucciones de nuestro lenguaje. En este caso, hemos elegido traducir las expresiones a código python, escribir el resultado en una variable y ejecutarlo.

Esta parte es, probablemente, la de menos interés, ya que únicamente consiste en recorrer una vez más el código generando un string que contenga el programa completo pero con otra sintaxis, por lo que no nos extenderemos mucho en la explicación. Cada clase consta ahora además de una función *Code*, similar a *Tipo* en la forma de ejecutarse, que realiza esta traducción. Sin embargo, cabe destacar el sistema de comprobación de errores.

Desde nuestro main cambiamos la salida estándar de la pantalla a un fichero e imprimimos en él el resultado de hacer *exec* sobre el string generado. Para hacer esto posible, hemos creado un fichero externo que contendrá la función *printf* (función para imprimir por pantalla en C) programada en python,y que se importará al comenzar cada programa. En este fichero se podrán seguir añadiendo funciones para ser llamadas desde nuestros programas C. Mediante nuestra nueva función, ya tenemos todo listo para comprobar que todo funciona bien.

Llegados a este punto, ya tenemos un prototipo de compilador de lenguaje C, con todas sus funcionalidades básicas implementadas.

6 Anexo

6.1 Código del Lexer

6.1.1 Tokens como expresiones regulares

```
# Sets of literals
LE = r'<='
GE = r'>='
EQUAL = r'=='
DIFFERENT = r'!='
PLUSEQ = r'\+='
MINUSEQ = r'-='
PLUSPLUS = r'\+\+'
MINUSMINUS = r'--'

# Keywords
BREAK = r'\bbreak\b'
CASE = r'\bcase\b'
CHAR = r'\bchar\b'
CONTINUE = r'\bcontinue\b'
DOUBLE = r'\bdouble\b'
ELSE = r'\belse\b'
FLOAT= r'\bfloat\b'
FOR = r'\bfor\b'
IF= r'\bif\b'
INT= r'\bint\b'
LONG= r'\blong\b'
RETURN= r'\breturn\b'
SWITCH= r'\bswitch\b'
VOID= r'\bvoid\b'
WHILE= r'\bwhile\b'

# Identifiers
IDENTIFIER = r'[a-zA-Z_]+[a-zA-Z_0-9]*'

# Type values
DECIMAL_CONST = r'-?[0-9]+\.[0-9]+'
INT_CONST = r'-?[0-9]+'
CHAR_CONST = r'\'.{1}\''
```

6.1.2 Lexer de Comentarios

```
class Comments(Lexer):
    _contador = 1

    tokens = {}

    @_(r'.$')
    def EOF(self, t):
        self.lineno += 1
        t.lineno += 1
        self._caracteres = ''
        t.type = "ERROR"
        t.value = "EOF in comment"
        return t


    @_(r'\*\)$')
    def FINFILE2(self, t):
        self._contador -= 1
        if self._contador == 0:
```

```
            self._contador = 1
            self.begin(CLexer)
        else:
            self._caracteres = ''
            t.type = "ERROR"
            t.value = "EOF in comment"
            return t


    @_(r'/\*')
    def BEGINING(self, t):
        self._contador += 1

    @_(r'\*/')
    def END(self, t):
        self._contador -= 1
        if self._contador == 0:
            self._contador = 1
            self.begin(CLexer)

    @_(r'\n')
    def NEW_LINE(self, t):
        self.lineno += 1

    @_(r'.')
    def CHARACTER(self, t):
        t.value = ''
```

6.1.3 Lexer de Strings

```
class Strings(Lexer):

    _caracteres = ''
    _null = False
    _nullEscapado = False
    _counter = 0

    tokens = {STR_CONST, ERROR}

    @_(r'\\\n')
    def SALTO_ESCAPADO(self,t):
        self.lineno += 1
        self._counter +=1
        t.type = "STR_CONST"
        self._caracteres += r'\n'


    @_(r'\\\t')
    def TABULADOR_ESCAPADO(self,t):
        t.type = "STR_CONST"
        self._caracteres += r'\t'


    @_(r'\\\x08')
    def ESPACIO_ESCAPADO(self,t):
        t.type = "STR_CONST"
        self._caracteres += r'\b'


    @_(r'\\\f')
```

```
def FORMFEED_ESCAPADO(self,t):
    t.type = "STR_CONST"
    self._caracteres += r'\f'


@_(r'\\\\')
def DOBLE_BARRA(self, t):
    self._counter +=1
    t.type = "STR_CONST"
    self._caracteres += r"\\"


@_(r'\\[tnbf"]')
def ESPECIALES(self, t):
    t.type = "STR_CONST"
    self._caracteres += '\\' + t.value[1]
    self._counter += 1


@_(r'[\x01-\x08]|\x0B|\x0D|\x0F|[\x10-\x19]|\x1A|\x1B|\x1C|\x1D|\x1E|\x1F')
def CONTROL(self, t):
    t.type = "STR_CONST"
    self._caracteres += '\\' + oct(ord(t.value)).replace('o','')


@_(r'\x0C')
def NUEVA_PAGINA(self, t):
    t.type = "STR_CONST"
    self._caracteres += r'\f'


@_(r'\\\x00')
def NULO_ESCAPADO(self,t):
    self._nullEscapado = True


@_(r'\x00')
def NULO(self,t):
    self._null = True


@_(r'\\.')
def POR_DEFECTO(self, t):
    t.type = "STR_CONST"
    self._caracteres += t.value[1]


@_(r'\t')
def TABULADOR(self, t):
    t.type = "STR_CONST"
    self._caracteres += r'\t'


@_(r'[^\"]\n$')
def SALTO_Y_EOF(self, t):
    return error_string_sin_terminar(self, t)


@_(r'(.\\?$)|(.\\"$)')
def EOF(self, t):
    if t.value == '"':
```

```
            # Comprobacion de caracteres nulos
            if self._null == True:
                return error_nulo(self, t)

            if self._nullEscapado == True:
                return error_nulo_escapado(self, t)

            # Correcto
            return string_correcto(self, t)

        # EOF
        return error_eof(self, t)

    @_(r'.')
    def CARACTER(self, t):
        if t.value == '"':

            # Comprobacion de caracteres nulos
            if self._null == True:
                return error_nulo(self, t)

            if self._nullEscapado == True:
                return error_nulo_escapado(self, t)

            # Correcto
            return string_correcto(self, t)

        self._counter += 1
        self._caracteres += t.value


    @_(r'\n')
    def SALTO_LINEA(self,t):

        # Comprobacion de caracteres nulos
        if self._null == True:
            return error_nulo(self, t)

        if self._nullEscapado == True:
            return error_nulo_escapado(self, t)

        return error_string_sin_terminar(self, t)
```

6.1.4 Lexer principal

```
class CLexer(Lexer):

    @_(r'//.*(\n|$)')
    def COMMENT(self, t):
        self.lineno += 1

    @_(r'/\*')
    def COMMENT_MULTILINE(self, t):
        self.begin(Comments)

    @_(r"\n")
    def NEW_LINE(self,t):
        self.lineno += 1

    @_(r'\'(\w|\d|\s)+\'')
```

```
    def ERROR(self, t):
        return t

    @_(r'\"')
    def STRINGS(self, t):
        self.begin(Strings)

    @_(r'\s')
    def BLANK(self, t):
        pass

    def salida(self, texto):
        list_strings = []
        lexer = CLexer()
        for token in lexer.tokenize(texto):
            result = f'#{token.lineno} {token.type} '
            if token.type == 'IDENTIFIER':
                if '[' in token.value:
                    result += f"{token.value[0:token.value.find('[')]}"
                else:
                    result += f"{token.value}"
            elif token.type in self.literals:
                result = f'#{token.lineno} \'{token.type}\''
            elif token.type == 'STR_CONST' or token.type == 'CHAR_CONST':
                result += token.value
            elif token.type == 'INT_CONST' or token.type == 'DECIMAL_CONST':
                result += str(token.value)
            elif token.type == 'STR_CONST':
                result += token.value
            elif token.type == 'ERROR':
                result = f'#{token.lineno} {token.type} "{token.value}"'
            else:
                result = f'#{token.lineno} {token.type}'

            list_strings.append(result)
        return list_strings
```

6.1.5 Gestión de errores

```
def error_nulo(self, t):
    if (t.value != '"'):
        self.lineno += 1
    self._counter = 0
    self._caracteres = ''
    self._null = False
    t.type = "ERROR"
    t.value = "String contains null character."
    self.begin(CLexer)
    return t


def error_nulo_escapado(self, t):
    if (t.value != '"'):
        self.lineno += 1
    self._counter = 0
    self._caracteres = ''
    self._nullEscapado = False
    t.type = "ERROR"
    t.value = "String contains escaped null character."
    self.begin(CLexer)
    return t
```

```
def error_eof(self, t):
    self._counter = 0
    self._caracteres = ''
    t.type = "ERROR"
    t.value = "EOF in string constant"
    self.begin(CLexer)
    return t


def error_string_sin_terminar(self, t):
    self.lineno += 1
    t.lineno += 1
    self._counter = 0
    self._caracteres = ''
    t.type = "ERROR"
    t.value = "Unterminated string constant"
    self.begin(CLexer)
    return t

def string_correcto(self, t):
    t.value = f'"{self._caracteres}"'
    t.type = "STR_CONST"
    self._counter = 0
    self._caracteres = ''
    self.begin(CLexer)
    return t
```

6.2 Código del Parser

6.2.1 Programa

```
#####################################   PROGRAMA   ####################################
@_('caracteristicas')
def program(self, p):
    return Programa(secuencia= p[0])
```

6.2.2 Característica

```
#####################################   CARACTERISTICA   ##################################
@_('')
def caracteristicas(self, p):
    return []

@_('caracteristicas caracteristica')
def caracteristicas(self, p):
    return p[0] + [p[1]]

@_('metodo')
def caracteristica(self, p):
    return p[0]

@_('atributo')
def caracteristica(self, p):
    return p[0]
```

6.2.3 Atributo

```
#####################################   ATRIBUTO   ####################################
@_('declaracion ";"')
def atributo(self, p):
    return Atributo(
        linea=p.lineno,
        tipo=p[0][0],
        nombre=p[0][1],
        cuerpo=NoExpr()
    )

@_('declaracion "=" expr ";"')
def atributo(self, p):
    return Atributo(
        linea=p.lineno,
        tipo=p[0][0],
        nombre=p[0][1],
        cuerpo=Asignacion(linea = p.lineno,
                          nombre=p[0][1],
                          cuerpo=p[2])
    )
```

6.2.4 Método

```
#####################################   METODO   ####################################
@_('declaracion "(" parametros ")" "{" exprs "}"')
def metodo(self, p):
    return Metodo(
        linea=p.lineno,
        tipo=p[0][0],
        nombre= p[0][1],
        formales=p[2],
        cuerpo=p[5]
    )
```

```
@_('declaracion "(" parametros ")" "{" "}"')
def metodo(self, p):
    return Metodo(
        linea=p.lineno,
        tipo=p[0][0],
        nombre= p[0][1],
        formales=p[2],
        cuerpo=NoExpr()
    )

@_('declaracion "(" parametros ")" ";"')
def metodo(self, p):
    return Metodo(
        linea=p.lineno,
        tipo=p[0][0],
        nombre= p[0][1],
        formales=p[2],
        cuerpo=NoExpr()
    )
```

6.2.5 Declaración y parámetros

```
######################################   DECLARACION   ######################################
@_('type IDENTIFIER')
def declaracion(self, p):
    return p[0], p[1]

@_('parametros "," parametro')
def parametros(self, p):
    return p[0] + [p[2]]

@_('parametro')
def parametros(self, p):
    return [p[0]]

@_('')
def parametros(self, p):
    return []

@_('declaracion')
def parametro(self, p):
    return Formal(
        linea= p.lineno,
        tipo = p[0][0],
        nombre_variable= p[0][1]
    )
```

6.2.6 Expresión

```
######################################   EXPR   ######################################
@_('exprs expr')
def exprs(self, p):
    return p[0] + [p[1]]

@_('exprs expr ";"')
def exprs(self, p):
    return p[0] + [p[1]]

@_('expr')
```

```
def exprs(self, p):
    return [p[0]]

@_('exprs_coma expr ","')
def exprs_coma(self, p):
    return p[0] + [p[1]]

@_('expr ","')
def exprs_coma(self, p):
    return [p[0]]

@_('declaracion ";"')
def expr(self, p):
    return Atributo(
        linea=p.lineno,
        tipo=p[0][0],
        nombre=p[0][1],
        cuerpo=NoExpr()
    )


@_('IDENTIFIER "=" expr ";"')
def expr(self, p):
    return Asignacion(
        p.lineno,
        nombre=p[0],
        cuerpo=p[2])

@_('declaracion "=" expr ";"')
def expr(self, p):
    return Atributo(
        linea=p.lineno,
        tipo=p[0][0],
        nombre=p[0][1],
        cuerpo=Asignacion(linea = p.lineno,
                          nombre=p[0][1],
                          cuerpo=p[2])
    )


@_('IDENTIFIER PLUSEQ expr')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Suma(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=p[2],
                    )
    )
@_('IDENTIFIER PLUSEQ expr ";"')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Suma(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=p[2],
                    )
    )
```

```
@_('IDENTIFIER MINUSEQ expr')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Resta(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=p[2],
                    )
    )
@_('IDENTIFIER MINUSEQ expr ";"')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Resta(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=p[2],
                    )
    )

@_('IDENTIFIER PLUSPLUS')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Suma(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=Entero(p.lineno, 1),
                    )
    )
@_('IDENTIFIER PLUSPLUS ";"')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Suma(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=Entero(p.lineno, 1),
                    )
    )

@_('IDENTIFIER MINUSMINUS')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Resta(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=Entero(p.lineno, 1),
                    )
    )
@_('IDENTIFIER MINUSMINUS ";"')
def expr(self, p):
    return Asignacion(
        linea = p.lineno,
        nombre=p[0],
        cuerpo=Resta(linea=p.lineno,
                    izquierda=Objeto(p.lineno, p[0]),
                    derecha=Entero(p.lineno, 1),
                    )
```

```
    )


@_('IDENTIFIER "(" exprs_coma expr ")"')
def expr(self, p):
    return LlamadaMetodo(p.lineno, nombre_metodo=p[0], argumentos=p[2] + [p[3]])


@_('IDENTIFIER "(" expr ")"')
def expr(self, p):
    return LlamadaMetodo(linea=p.lineno, nombre_metodo=p[0], argumentos = [p[2]])

@_('IDENTIFIER "(" ")"')
def expr(self, p):
    return LlamadaMetodo(linea=p.lineno, nombre_metodo=p[0])


@_('IF "(" expr ")" "{" exprs "}" elseifs')
def expr(self, p):
    return Condicional(
        linea=p.lineno,
        condicion=p[2],
        verdadero=p[5],
        falso=p[7]
    )

@_('elseifs elseif')
def elseifs(self, p):
    return p[0] + [p[1]]

@_('elseif')
def elseifs(self, p):
    return [p[0]]

@_('els')
def elseifs(self, p):
    return p[0]

@_('')
def elseifs(self, p):
    return []

@_('ELSE IF "(" expr ")" "{" exprs "}" elseifs')
def elseif(self, p):
    return Condicional(
        linea=p.lineno,
        condicion=p[3],
        verdadero=p[6],
        falso=p[8]
    )

@_('ELSE "{" exprs "}"')
def els(self, p):
    return p[2]

@_('')
def els(self, p):
    return []

@_('WHILE "(" expr ")" "{" exprs "}"')
def expr(self, p):
```

```
        return BucleWhile(
            linea=p.lineno,
            condicion=p[2],
            cuerpo=p[5]
        )

    @_('FOR "(" expr expr ";" expr ")" "{" exprs "}"')
    def expr(self, p):
        return BucleFor(
            linea=p.lineno,
            iter=p[2],
            condicion=p[3],
            accion=p[5],
            cuerpo=p[8]
            )

    @_('SWITCH "(" expr ")" "{" tiposCase "}"')
    def expr(self, p):
        return Swicht(
            linea=p.lineno,
            expr=p[2],
            casos=p[5]
        )

    @_('tiposCase tipoCase')
    def tiposCase(self, p):
        return p[0] + [p[1]]

    @_('tipoCase')
    def tiposCase(self, p):
        return [p[0]]

    @_('CASE expr ":" exprs')
    def tipoCase(self,p):
        return RamaCase(
            linea=p.lineno,
            valor=p[1],
            cuerpo=p[3]
        )

    @_('expr "+" expr')
    def expr(self, p):
        return Suma(p.lineno, p[0], p[2])

    @_('expr "-" expr')
    def expr(self, p):
        return Resta(p.lineno, p[0], p[2])

    @_('expr "*" expr')
    def expr(self, p):
        return Multiplicacion(p.lineno, p[0], p[2])

    @_('expr "/" expr')
    def expr(self, p):
        return Division(p.lineno, p[0], p[2])

    @_('expr "%" expr')
    def expr(self, p):
        return Modulo(p.lineno, p[0], p[2])

    @_('expr "<" expr')
```

```
def expr(self, p):
    return Menor(p.lineno, p[0], p[2])

@_('expr ">" expr')
def expr(self, p):
    return Mayor(p.lineno, p[0], p[2])

@_('expr LE expr')
def expr(self, p):
    return MenorIgual(p.lineno, p[0], p[2])

@_('expr GE expr')
def expr(self, p):
    return MayorIgual(p.lineno, p[0], p[2])

@_('expr EQUAL expr')
def expr(self, p):
    return Igual(p.lineno, p[0], p[2])

@_('expr DIFFERENT expr')
def expr(self, p):
    return Distinto(p.lineno, p[0], p[2])

@_('"!" expr')
def expr(self, p):
    return Not(p.lineno, p[1])

@_('"(" expr ")"')
def expr(self, p):
    return p[1]

@_('RETURN expr ";"')
def expr(self, p):
    return Retorno(
        linea=p.lineno,
        cuerpo=p[1]
    )

@_('BREAK ";"')
def expr(self, p):
    return Break(linea=p.lineno)

@_('CONTINUE ";"')
def expr(self, p):
    return Continue(linea=p.lineno)

@_('IDENTIFIER')
def expr(self, p):
    return Objeto(p.lineno, p[0])

@_('INT_CONST')
def expr(self, p):
    return Entero(p.lineno, p[0])

@_('CHAR_CONST')
def expr(self, p):
    return Char(p.lineno, p[0])

@_('DECIMAL_CONST')
def expr(self, p):
    return Decimal(p.lineno, p[0])
```

```
@_('STR_CONST')
def expr(self, p):
    return String(p.lineno, p[0])
```

6.2.7 Tipo

```
#####################################   TIPO   ######################################
@_('INT', 'CHAR', 'VOID')
def type(self, p):
    return p[0]

@_('FLOAT', 'DOUBLE', 'LONG')
def type(self, p):
    return "float"
```

6.3 Código del Analizador Semántico e Intérprete

6.3.1 Clase Ambito

```
def enterScope(self):
    if len(self.pila) == 0:
        self.pila.append(copy.copy({}))
    else:
        self.pila.append(copy.copy(self.pila[-1]))

def findSymbol(self, nombre):
    return self.pila[-1][nombre]

def addSymbol(self, x):
    if isinstance(x, Formal):
        self.pila[-1][x.nombre_variable] = x.tipo
    else:
        self.pila[-1][x.nombre] = x.tipo

def checkScope(self, x):
    if x in self.pila[-1].keys():
        return True
    else:
        return False

def exitScope(self):
    self.pila.pop()
```

6.3.2 Clase Programa

```
class Programa(IterableNodo):
    def code(self, n):
        codigo = ""
        for caracteristica in self.secuencia:
            codigo += caracteristica.code(n)
        return codigo

    def Tipo(self):

        amb = Ambito()
        amb.enterScope()
        tmpAttr = []
        tmpMet = []
        for caracteristica in self.secuencia:

            # Si es metodo
            if isinstance(caracteristica, Metodo):
                args = []
                for arg in caracteristica.formales:
                    if arg.nombre_variable not in list(map(itemgetter(0), args)):
                        args.append([arg.nombre_variable, arg.tipo])
                    # Argumento redefinido
                    else:
                        raise Exception(f'{arg.linea}: Formal parameter \
                        {arg.nombre_variable} is multiply defined.')
                tmpMet.append(caracteristica.nombre)
                amb.metodos[caracteristica.nombre] = [args, caracteristica.tipo]

            # Si es atributo
            else:
                tmpAttr.append(caracteristica.nombre)
                amb.addSymbol(caracteristica)
```

```
        # Comprobar si falta el main
        if "main" not in amb.metodos:
            raise Exception('Method main is not defined.')

        # Comprobar metodos redefinidos
        dup = {x for x in tmpMet if tmpMet.count(x) > 1}
        if len(dup) > 0:
            raise Exception(f'Method {dup[0]} defined multiple times.')

        # Comprobar atributos globales redefinidos
        dup = {x for x in tmpAttr if tmpAttr.count(x) > 1}
        if len(dup) > 0:
            raise Exception(f'Attribute {dup[0]} defined multiple times.')

        for caracteristica in self.secuencia:
            caracteristica.Tipo(amb)

        amb.exitScope()
        amb.reset()
```

6.3.3 Clase Atributo

```
class Atributo(Caracteristica):
    def code(self, n):
        codigo = ''

        if isinstance(self.cuerpo, NoExpr):
            if self.tipo == 'int':
                codigo += f'{" "*n}{self.nombre}: {self.tipo} = 0'
            elif self.tipo == 'string' or self.tipo == 'char':
                codigo += f'{" "*n}{self.nombre}: str = ""'
            elif self.tipo == 'float':
                codigo += f'{" "*n}{self.nombre}: {self.tipo} = 0.0'
            else:
                codigo += f'{" "*n}{self.nombre}: {self.tipo} = None'

        elif isinstance(self.cuerpo, Asignacion):
            codigo += f'{" "*n}{self.nombre}: {self.tipo} = {self.cuerpo.cuerpo.code(0)}'

        else:
            codigo += f'{" "*n}{self.nombre}: {self.tipo} = {self.cuerpo.code(0)}'

        return codigo

    def Tipo(self, Ambito):

        self.cuerpo.Tipo(Ambito)

        if isinstance(self.cuerpo, Asignacion):
            if self.cuerpo.cast != self.tipo:
                raise Exception(f'{self.linea}: Inferred return type {self.cuerpo.cast} \
                 of {self.nombre} does not conform to declared type {self.tipo}.')
```

6.3.4 Clase Metodo

```
class Metodo(Caracteristica):
    def code(self, n):
        codigo = f'{" "*n}def {self.nombre}('
        for i in range(0, len(self.formales)):
            codigo += self.formales[i].code(0)
```

```
            if i != (len(self.formales) - 1):
                codigo += f', '
        codigo += f'):\n'

        for expr in self.cuerpo:
            codigo += expr.code(n+2)
            codigo += ('\n')

        return codigo

    def Tipo(self, Ambito):

        # Anhadir argumentos al ambito
        Ambito.enterScope()
        for formal in self.formales:
            Ambito.addSymbol(formal)

        # Computar tipos de las expresiones y buscar tipos de retorno
        returnTypes = []
        if not isinstance(self.cuerpo, NoExpr):
            for expr in self.cuerpo:
                if isinstance(expr, Atributo):
                    Ambito.addSymbol(expr)

                expr.Tipo(Ambito)

                if isinstance(expr, Retorno):
                    returnTypes.append(expr.cast)
        else:
            self.cuerpo.cast = "void"

        if len(returnTypes) == 0:
            returnTypes.append("void")

        # Comprobar que el tipo de retorno coincide con el tipo de lo que retorna el cuerpo
        for tipo in returnTypes:
            if tipo != self.tipo:
                raise Exception(f'{self.linea}: Inferred return type {tipo} of \
                                    method {self.nombre} does not conform to \
                                    declared return type {self.tipo}.')

        Ambito.exitScope()
```

6.3.5 Expresiones Aritméticas

```
class OperacionBinaria(Expresion):
    izquierda: Expresion = None
    derecha: Expresion = None

    def code(self, n):
        codigo = f'{" "*n}{self.izquierda.code(0)} {self.operando} {self.derecha.code(0)}'
        return codigo


class Suma(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.derecha.cast == 'int' or self.derecha.cast == 'float'):
            self.cast = self.derecha.cast
```

```
        else:
            raise Exception(f'{self.linea}: non-numerical arguments: \
            {self.izquierda.cast} + {self.derecha.cast}')


class Resta(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.derecha.cast == 'int' or self.derecha.cast == 'float'):
            self.cast = self.derecha.cast
        else:
            raise Exception(f'{self.linea}: non-numerical arguments: \
            {self.izquierda.cast} + {self.derecha.cast}')


class Multiplicacion(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.derecha.cast == 'int' or self.derecha.cast == 'float'):
            self.cast = self.derecha.cast
        else:
            raise Exception(f'{self.linea}: non-numerical arguments: \
            {self.izquierda.cast} + {self.derecha.cast}')


class Division(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.derecha.cast == 'int' or self.derecha.cast == 'float'):
            self.cast = self.derecha.cast
        else:
            raise Exception(f'{self.linea}: non-numerical arguments: \
            {self.izquierda.cast} + {self.derecha.cast}')


class Modulo(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.derecha.cast == 'int'):
            self.cast = self.derecha.cast
        else:
            raise Exception(f'{self.linea}: non-numerical arguments: \
            {self.izquierda.cast} + {self.derecha.cast}')
```

6.3.6 Expresiones Relacionales y Lógicas

```
class Mayor(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.izquierda.cast == 'int' or self.izquierda.cast == 'float'):
            self.cast = 'bool'
```

```
        else:
            raise Exception(f'{self.linea}: Error in > statement.')


class MayorIgual(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.izquierda.cast == 'int' or self.izquierda.cast == 'float'):
            self.cast = 'bool'
        else:
            raise Exception(f'{self.linea}: Error in >= statement.')


class Menor(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.izquierda.cast == 'int' or self.izquierda.cast == 'float'):
            self.cast = 'bool'
        else:
            raise Exception(f'{self.linea}: Error in < statement.')


class MenorIgual(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if (self.izquierda.cast == self.derecha.cast) and
        (self.izquierda.cast == 'int' or self.izquierda.cast == 'float'):
            self.cast = 'bool'
        else:
            raise Exception(f'{self.linea}: Error in <= statement.')


class Igual(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if self.izquierda.cast == self.derecha.cast:
            self.cast = 'bool'
        else:
            raise Exception(f'{self.linea}: Error in == statement.')


class Distinto(OperacionBinaria):
    def Tipo(self, Ambito):
        self.izquierda.Tipo(Ambito)
        self.derecha.Tipo(Ambito)
        if self.izquierda.cast == self.derecha.cast:
            self.cast = 'bool'
        else:
            raise Exception(f'{self.linea}: Error in != statement.')


class Not(Expresion):
    def Tipo(self, Ambito):
        self.expr.Tipo(Ambito)
        if (self.expr.cast == "bool") or (self.expr.cast == "int"):
```

```
            self.cast = "Bool"
        else:
            raise Exception(f'{self.linea}: Error in negation statement.')
```

6.3.7 Sentencias If, Switch, While y For

```
class Condicional(Expresion):
    def code(self, n):
        codigo = f'{" "*n}if {self.condicion.code(0)} :\n'
        for expr in self.verdadero:
            codigo += f'{" "*(n+2)}{expr.code(0)}\n'

        for e in self.falso:
            if isinstance(e, Condicional):
                codigo += f'{" "*n}elif ({e.condicion.code(0)}):\n'
                for expr in e.verdadero:
                    codigo += f'{" "*(n+2)}{expr.code(0)}\n'
                codigo += f'{" "*n}else:\n'
                if e.falso == []:
                    codigo += f'{" "*(n+2)}pass\n'
                for expr in e.falso:
                    codigo += f'{" "*(n+2)}{expr.code(0)} \n'
            else:
                codigo += f'{" "*n}else:\n'
                for expr in e:
                    codigo += f'{" "*(n+2)}{expr.code(0)} \n'

        if self.falso == []:
            codigo += f'{" "*(n+2)}pass\n'

        return codigo

    def Tipo(self, Ambito):
        Ambito.enterScope()
        self.condicion.Tipo(Ambito)

        for expr in self.verdadero:
            if isinstance(expr, Atributo):
                Ambito.addSymbol(expr)
            expr.Tipo(Ambito)

        for expr in self.falso:
            if isinstance(expr, Atributo):
                Ambito.addSymbol(expr)
            expr.Tipo(Ambito)

        self.cast = "void"
        Ambito.exitScope()


class BucleWhile(Expresion):
    def code(self, n):
        codigo = f'{" "*n}while {self.condicion.code(0)}:\n'
        for expr in self.cuerpo:
            codigo += expr.code(n+2) + '\n'
        return codigo

    def Tipo(self, Ambito):
        Ambito.enterScope()
```

```
        self.condicion.Tipo(Ambito)
        if self.condicion.cast != 'bool':
            raise Exception(f'{self.linea}: Loop condition does not have type bool.')

        for expr in self.cuerpo:
            if isinstance(expr, Atributo):
                Ambito.addSymbol(expr)
            expr.Tipo(Ambito)

        self.cast = 'void'
        Ambito.exitScope()


class BucleFor(Expresion):
    def code(self, n):
        codigo = f'{" "*n}{self.iter.code(0)}\n'
        codigo += f'{" "*n}while {self.condicion.code(0)}:\n'
        for expr in self.cuerpo:
            codigo += expr.code(n+2) + '\n'
        codigo += self.accion.code(n+2) + '\n'
        return codigo

    def Tipo(self, Ambito):

        Ambito.enterScope()
        if isinstance(self.iter, Atributo):
            Ambito.addSymbol(self.iter)

        self.iter.Tipo(Ambito)
        self.condicion.Tipo(Ambito)
        self.accion.Tipo(Ambito)

        if self.condicion.cast != 'bool':
            raise Exception(f'{self.linea}: Loop condition does not have type bool.')

        for expr in self.cuerpo:
            if isinstance(expr, Atributo):
                Ambito.addSymbol(expr)
            expr.Tipo(Ambito)

        self.cast = 'void'

        Ambito.exitScope()


class RamaCase(Expresion):
    def code(self, n):
        codigo = f'{" "*n}case {self.valor.code(0)}:\n'
        for expr in self.cuerpo:
            if not isinstance(expr, Break):
                codigo += expr.code(n+2) + '\n'
        return codigo

    def Tipo(self, Ambito):
        self.valor.Tipo(Ambito)

        Ambito.enterScope()
        for expr in self.cuerpo:
            if isinstance(expr, Atributo):
                Ambito.addSymbol(expr)
            expr.Tipo(Ambito)
```

```
        Ambito.exitScope()

        self.cast = 'void'


class Swicht(Expresion):
    def code(self, n):
        codigo = f'{" "*n}match {self.expr.code(0)}:\n'
        for expr in self.casos:
            codigo += expr.code(n+2) + '\n'
        return codigo

    def Tipo(self, Ambito):
        self.expr.Tipo(Ambito)
        for caso in self.casos:
            caso.Tipo(Ambito)
        self.cast = 'void'
```

6.3.8 Clase Asignacion

```
class Asignacion(Expresion):
    def code(self, n):
        codigo = f'{" "*n}{self.nombre} = {self.cuerpo.code(0)}'
        return codigo

    def Tipo(self, Ambito):
        tipoNombre = Ambito.tipoAtributo(self.nombre)
        self.cuerpo.Tipo(Ambito)
        tipoCuerpo = self.cuerpo.cast

        # Comprobacion de tipos
        if tipoCuerpo != tipoNombre:
            raise Exception(f'{self.linea}: Type {tipoCuerpo} of assigned \
                            expression does not conform to declared type {tipoNombre} \
                            of identifier {self.nombre}.')
        else:
            self.cast = tipoNombre
```

6.3.9 Clase LlamadaMetodo

```
class LlamadaMetodo(Expresion):
    def code(self, n):
        codigo = ""
        codigo += f'{" "*n}{self.nombre_metodo}('
        for i in range(0, len(self.argumentos)):
            if isinstance(self.argumentos[i], Objeto):
                codigo += f'{self.argumentos[i].code(0)}'
            else:
                codigo += f'{self.argumentos[i].code(0)}'
            if i != (len(self.argumentos) - 1):
                codigo += f', '
        codigo += f')'

        return codigo

    def Tipo(self, Ambito):

        # Comprobar si el metodo esta definido
        c = Ambito.checkMetodo(self.nombre_metodo)
        if c != False:
            metodo = Ambito.metodos[self.nombre_metodo]
```

```
        else:
            raise Exception(f'{self.linea}: Dispatch to undefined method {self.nombre_metodo}.')
        # Comprobacion del tipo de los argumentos
        for i in range(len(self.argumentos)):
            self.argumentos[i].Tipo(Ambito)
            if self.argumentos[i].cast != metodo[0][i][1] and metodo[0][i][1] != '':
                raise Exception(f'{self.linea}: In call of method {self.nombre_metodo}, \
                                type {self.argumentos[i].cast} of parameter \
                                {metodo[0][i][0]} does not conform to declared \
                                type {metodo[0][i][1]}.')

        # Tipo de retorno
        self.cast = metodo[1]
```

6.3.10 Clase Objeto

```
class Objeto(Expresion):
    def code(self, n):
        codigo = f'{" "*n}{self.nombre}'
        return codigo

    def Tipo(self, Ambito):

        if Ambito.checkScope(self.nombre):
            self.cast = Ambito.findSymbol(self.nombre)
        elif Ambito.checkAtributo(self.nombre):
            self.cast = Ambito.tipoAtributo(self.nombre)
        else:
            raise Exception(f'{self.linea}: Undeclared identifier {self.nombre}.')
```

6.3.11 Tipos Básicos

```
class Entero(Expresion):
    def code(self, n):
        codigo = f'{" "*n}{self.valor}'
        return codigo

    def Tipo(self, Ambito):
        self.cast = "int"


class Decimal(Expresion):
    def code(self, n):
        codigo = f'{" "*n}{self.valor}'
        return codigo

    def Tipo(self, Ambito):
        self.cast = "float"


class Char(Expresion):
    def code(self, n):
        v = self.valor[1:-1]
        codigo = f'{" "*n}"{v}"'
        return codigo

    def Tipo(self, Ambito):
        self.cast = "char"


class String(Expresion):
```

```
    def code(self, n):
        codigo = f'{" "*n}{self.valor}'
        return codigo

    def Tipo(self, Ambito):
        self.cast = "string"
```

6.3.12 Return, Break y Continue

```
class Retorno(Expresion):
    def code(self, n):
        codigo = f'{" "*n}return {self.cuerpo.code(0)}'
        return codigo

    def Tipo(self, Ambito):
        self.cuerpo.Tipo(Ambito)
        self.cast = self.cuerpo.cast


class Break(Expresion):
    def code(self, n):
        codigo = f'{" "*n}break'
        return codigo

    def Tipo(self, Ambito):
        self.cast = 'void'


class Continue(Expresion):
    def code(self, n):
        codigo = f'{" "*n}continue'
        return codigo

    def Tipo(self, Ambito):
        self.cast = 'void'
```