MEMORIA COMPILADORES

ANALIZADOR SINTÁCTICO y SEMÁNTICO

GRUPO 8

Luis Ildefonso Gómez Solana 040078
Fernando Arias Porras 030035
Víctor Gómez Aragoneses 040071

Índice de contenido

INTRODUCCIÓN

Proyecto

La práctica asignada para el presente curso es un subconjunto del lenguaje C++. Las características principales que podemos denotar y que son parte común de la práctica son la declaración de clases, métodos, variables globales o locales y *main*. De igual manera tendremos operaciones de entrada-salida, llamadas a métodos, expresiones.
Concretando, la parte común de nuestro proyecto es:

- Definición y uso de clases, con atributos y métodos.
- Definición de funciones.
- Tipos enteros, lógicos y vacíos.
- Variables enteras y su declaración.
- Constantes enteras y cadenas de caracteres (entre comillas dobles).
- Sentencias: asignación, condicional simple, llamada a funciones, métodos y retorno.
- Expresiones.
- Comentarios.
- Operaciones de entrada/salida por terminal (*cin*, *cout*).
- Operadores.

Por otra parte, la parte específica para nuestro proyecto serán las operaciones condicionales complejas (*if-then-else*), parámetros por valor, asignación con operación (p.e. +=) y en cuanto al analizador sintáctico: descendente recursivo. De igual manera que en el caso anterior podemos verlo más claramente así:

- Sentencias: Sentencia condicional (if, if-else)
- Parámetros: Por valor.
- Operador especial: Asignación con operación (op=)
- Analizador Sintáctico: Descenso recursivo.

Para la realización del proyecto decidimos usar una herramienta que cumpliera en gran medida todo lo que se exigía y que no provocara excesivos problemas a la hora de generar los módulos del compilador. En concreto, usamos **ANother Tool for Language Recognition** (ANTLR) que genera un analizador sintáctico LL(k), a parte que dicha herramienta tiene gran cantidad de documentación publicada.

ANALIZADOR SINTÁCTICO

El analizador sintáctico se encargará de transformar la salida del léxico (con los sucesivos *tokens*) en un árbol de derivación que más adelante será el que reciba el semántico.

En nuestro caso el analizador sintáctico será descendente recursivo, ya que, nuestra herramienta (ANTLR) genera dicho árbol. En éste analizador las entradas son de izquierda a la derecha, y construcciones de derivaciones de derivaciones por la izquierda de una sentencia. Este tipo de gramáticas son llamadas gramáticas LL.

Básicamente nuestro analizador sintáctico se encargará de pedir al analizador léxico *tokens* y comprobará si puede generar una regla sintáctica con ellos en otro caso generará un error y lo imprimirá por pantalla.

Gramática del lenguaje:

A continuación redactamos las reglas sintácticas que componen nuestra gramática. El axioma es “programa”, y los tokens provenientes del análisis léxico se expresan en mayúscula:

programa ::= ant_prog siguiente_prog
ant_prog ::= instDecVar ant_prog | λ
siguiente_prog ::= decMetodo siguiente_prog | subprograma siguiente_prog | decClase siguiente_prog | main
main ::= ttipo MAIN PARENT_AB listaDecParams PARENT_CE LLAVE_AB cuerpo_sp LLAVE_CE
instDecVar ::= listaDeclaraciones PUNTO_COMA
listaDeclaraciones ::= ttipo declaracion siguiente_declaracion
siguiente_declaracion ::= COMA declaracion siguiente_declaracion | λ
declaracion ::= IDENT | IDENT OP_ASIG q_argumento | IDENT PARENT_AB q_argumento PARENT_CE | IDENT CORCHETE_AB intdec CORCHETE_CE
intdec ::= LIT_ENTERO_OCTAL | LIT_ENTERO_DECIMAL
subprograma ::= ttipo IDENT PARENT_AB listaDecParams PARENT_CE
q_argumento ::= LIT_CADENA | LIT_ENTERO_OCTAL | LIT_ENTERO_DECIMAL | IDENT PARENT_AB q_argumento PARENT_CE | IDENT | CTE_LOGTRUE | CTE_LOGFALSE | λ
ttipo ::= INT | BOOL | VOID | CHAR | IDENT
decMetodo ::= ttipo IDENT DOSPUNTOS_DOS IDENT PARENT_AB listaDecParams PARENT_CE LLAVE_AB cuerpo_sp LLACE_CE
listaDecParams ::= ttipo | listaDeclaraciones siguiente_listaDecParams | λ
siguiente_listaDecParams ::= COMA listaDeclaraciones siguiente_listaDecParams | λ
decClase ::= CLASS IDENT LLAVE_AB cuerpo_clase parte_publica parte_privada LLAVE_CE PUNTO_COMA
cuerpo_clase ::= ttipo IDENT comun_clase PUNTO_COMA cuerpo_clase | λ
comun_clase ::= COMA IDENT comun_clase_sig1 | PARENT_AB ttipo comun_clase_sig2 PARENT_CE
comun_clase_sig1 ::= COMA IDENT comun_clase_sig1 | λ
comun_clase_sig2 ::= COMA ttipo | λ
parte_publica ::= PUBLIC DOSPUNTOS cuerpo_clase | λ
parte_privada ::= PRIVATE DOSPUNTOS cuerpo_clase | λ
cuerpo_sp ::= instrucción cuerpo_sp | λ
instruccion ::= instDecVar | instExpresion | instNula | instCond | instCout | instCin | instReturn
instReturn ::= RETURN PUNTO_COMA | RETURN instExpresion
instExpresion ::= expresion PUNTO_COMA
expresion ::= expAsignacion
expAsignacion ::= expOLogico | expOLogico expAsig_sig
expAsig_sig ::= OP_ASIG instCondSimple | OP_ASIG expOLogico | OP_ASIG_MAS instCondSimple | OP_ASIG_MAS expOLogico
expOLogico ::= expYLogico | OP_OR expYLogico
expYLogico ::= expComparacion | expComparacion OP_AND expComparacion
expComparacion ::= expAritmetica ops expAritmetica expComparacion | λ
ops ::= OP_IGUAL | OP_DISTINTO | OP_MAYOR | OP_MENOR | OP_MENOR_IGUAL | OP_MAYOR_IGUAL
expAritmetica ::= expProducto expAritmetica_siguiente

expAritmetica_siguiente ::= opsmasmenos expProducto expAritmetica_siguiente | λ
opsmasmenos ::= OP_MAS | OP_MENOS
expProducto ::= expCambioSigno expProducto_siguiente
expProducto_siguiente ::= OP_PRODUCTO expCambioSigno | OP_DIVISION expCambioSigno | λ
expCambioSigno ::= OP_MENOS expPostIncremento | OP_MAS expPostIncremento | expPostIncremento
expPostIncremento ::= expNegacion OP_MASMAS | expNegacion OP_MENOSMENOS
expNegacion ::= expNegacion_sig acceso
expNegacion_sig ::= OP_NOT expNegacion_sig | λ
acceso ::= raizAcceso acceso_sig | raizAccesoConSubAccesos acceso_sig | literal | llamada | IDENT | raizAccesoSinAccesos | raizAccesoSinAccesos acceso_sig
acceso_sig::= PUNTO subAcceso acceso_sig | λ
raizAcceso ::= IDENT | literal
raizAccesoConSubAccesos ::= OP_MAS
raizAccesoSinAcceso ::= llamada
subAcceso ::= IDENT | llamada
llamada ::= IDENT PARENT_AB listaExpresiones PARENT_CE
listaExpresiones ::= IDENT listaExpresiones_sig | literal listaExpresiones_sig | λ
listaExpresiones_sig ::= COMA IDENT lista_expresiones_sig | COMA literal lista_expresiones_sig | λ
literal ::= LIT_ENTERO_OCTAL | LIT_ENTERO_DECIMAL | LIT_CADENA | CTE_LOGTRUE | CTE_LOGFALSE
instCond ::= IF PARENT_AB expresion PARENT_CE LLAVE_AB cuerpo_sp LLAVE_CE sino | IF PARENT_AB expresion PARENT_CE instrucción sino
sino ::= sinosi sino | sinofin
sinosi ::= ELSE IF PARENT_AB expresion PARENT_CE LLAVE_AB cuerpo_sp LLAVE_CE | ELSE IF PARENT_AB expresion PARENT_CE instrucción
sinofin ::= ELSE LLAVE_AB! cuerpo_sp LLAVE_CE! | ELSE instrucción
instNula ::= PUNTO_COMA
instDecMet ::= IDENT PUNTO IDENT PARENT_AB lista_valores PARENT_CE PUNTO_COMA
lista_valores ::= q_argumento lista_valores_sig
lista_valores_sig ::= COMA q_argumento lista_valores_sig | λ
e_vector ::= IDENT CORCHETE_AB LIT_ENTERO_DECIMAL CORCHETE_CE
cadena ::= vector
vector ::= CORCHETE_AB LIT_ENTERO_DECIMAL CORCHETE_CE
instCout ::= COUT MENOR_MENOR arg_io instCout_sig PUNTO_COMA
instCout_sig ::= MENOR_MENOR arg_io instCout_sig | λ
instCin ::= CIN MAYOR_MAYOR arg_io PUNTO_COMA
arg_io ::= IDENT | LIT_CADENA
instCondSimple ::= PARENT_AB acceso op_cond acceso PARENT_CE INTER acceso DOSPUNTOS acceso
op_cond ::= OP_IGUAL | OP_DISTINTO | OP_MAYOR OP_MENOR | OP_MENOR_IGUAL | OP_MAYOR_IGUAL

Errores detectados:

A continuación relatamos los diversos errores que el analizador sintáctico detecta, alguno de ellos son genéricos puesto que la falta de un *token* para completar una regla sintáctica o el error en la terminación de la regla es muy común.

Nuestro analizador sintáctico, tal y como lo hemos construido, debe consumir tres *tokens* para poder determinar un único camino que seguir, por ejemplo: "*int variable;*" produce una variable y sólo se puede producir este elemento, en otro caso tendríamos una gramática ambigua (no sería determinista).

Con respecto a lo anterior, el analizador sintáctico generará un error cuando falle en la generación de una regla, generando un mensaje del tipo "Se esperaba el *token* %t y se recibió un %o", entendiendo por %t el *token* que se esperaba recibir para completar y continuar con el análisis y por %o el *token* que se recibió y genero el error.
En la generación de cabeceras de programas, métodos y *main* se puede producir un error en al declaración de los parámetros de entrada, en este caso se emitirá un error del tipo "Lista de parámetros es inválida", también puede ocurrir que una llamada a un método produzca error,emitiendo por ello el mismo error. De igual manera cuando una expresión falle se emitirá un mensaje "Expresión inválida".

Por último, si el código carece de la función *main* el analizador sintáctico generará un error recordándonos la necesidad de declararlo.

ANALIZADOR SEMÁNTICO

Diseño

La fase de análisis semántico de un procesador de lenguaje es aquélla que computa la información adicional necesaria para el procesamiento de un lenguaje, una vez que la estructura sintáctica de un programa haya sido obtenida. Es por tanto la fase posterior a la de análisis sintáctico y la última dentro del proceso de síntesis de un lenguaje de programación.

La finalidad de dicho análisis es dar validez semántica a las sentencias aceptadas por el analizador sintáctico.

Una de los puntos más importantes de este módulo es la comprobación y verificación de tipos. Por ejemplo, en nuestro caso, no podremos asignar a una variable tipo cadena la palabra reservada *true*, ya que, no pertenece a dicho conjunto de valores posibles.

Errores detectados:

La generación de errores semánticos hacen especial énfasis en el correcto uso de los tipos. Esto se ve por ejemplo en asignación errónea de valores a variables que no son del mismo tipo de dicho elemento. Para ello se generará un mensaje que nos advierta del error. Por otra parte tenemos las llamadas a métodos y funciones que provocarán, al igual que el caso anterior, un error de tipos; en este caso el error nos advertirá del problema de tipos que hemos cometido.

Por otra parte las expresiones, podemos advertir el error de tipos que se puede dar al intentar operar con tipos diferentes o incluso con cadena de caracteres, esto será un error como es lógico.

Por último tenemos la declaración de programas o métodos pertenecientes a clases que no existen o el intento de declaración de variables o llamadas con el mismo nombre que otro elemento, todo ello provocará un error advirtiéndonos que ya se ha definido previamente esa declaración o por el contrario que no lo está.

CASOS DE PRUEBAS

A continuación se detallan el código de los programas de pruebas que evalúan los diferentes casos de éxito o error de nuestro compilador.

Pruebas correctas.

Prueba 1:

```
// Prueba Correcta
//

// Inclusion de variables globales
int hola, var_global;
int adios=90;  // inicializadas
char *s;

void demo ()   // funcion sin argumentos
{
                // No hace nada y
        return; // No devuelve nada
}

void main (int variable) {
        // Declaracion de variables
        // algunas inicializadas.
        int otra_var, otra_var2;
        int otra_var2=20;

        return otra_var;
}
```

Prueba 2:

```
// Prueba Correcta

// Declaracion de subprograma o funcion
int funcionAnodina (int n) {
        int uno = 1, cero = 0; // variables locales inicializadas

        // intrucciones operacionales
        uno = cero * 90;
        uno = uno + 34 - 23 / 30 - cero * n;

        return uno;
}

// Punto ppal del programa
int main (int n) {
        int variable;
        // Asignacion de valor cierto ó falso
        variable = n * funcionAnodina(n);

        return variable;
}
```

Prueba 3:

```
// Prueba Correcta

// declaracion de metodo y clase
//Declaración de la clase Fecha:
class Fecha {
public: int a, b, c;
        int daDia (void);       // método que devuelve un entero
private:
        int d,m,a;      // tres enteros privados
};

// Definicion de un metodo daDia de la clase Fecha
int Fecha::daDia (void){
        return d;
}

void main (int dia) {
        Fecha var;
        return var.daDia();
}
```

Prueba 4:

```
// Prueba Correcta

// main
bool main (int n, int m)
{
        /* Declaracion de variables
         - algunas inicializadas - */
        bool variable = true;

        // Condicional Complejo
        if (variable)    // cierto
                variable = false;
        else
                varible = true;

        // Devolvemos le valor calculado
        return variable;
}
```

Prueba 5:

```
// Prueba Correcta

// Objetivo:
// Comprobamos el correcto funcionamiento del
// condicional complejo con varias posibilidades
// y la asignacion con operacion.


// main
void main (void) {
        int num = 30;
        int algo =40, otro;

        if (num == 0) {
                otro += algo;
        }
        else if (algo == 0 || otro == 0) {
                otro += algo;
        }

        if (num==30) {
                num = num - num;
        } else {
                num = 30;
        }
}
```

Prueba6:

```
// Prueba correcta

/* Objetivo
        Comprobamos el correcto funcionamiento
de la declaracion de programas/funciones. Así
como las operaciones de cin y cout
*/

void imprime (char *msg,  f) {
        cout << s << msg << f;
        cout << "\n";  // imprime un salto de línea */
        return;
}

void main (void) {
        char* s="hola Mundo!";
        cout << s;
        cin >> s;
        return;
}
```

Pruebas incorrectas

Prueba 1:

```
// Prueba Incorrecta
//

// Inclusion de variables globales
int hola, var_global,;   // error en la coma
int adios=90;  // inicializadas
char *s          // falta cerrar la declaracion.

void demo ()   // funcion sin argumentos
{
                  // No hace nada y
        return; // No devuelve nada
}

void main (int variable) {
        // Declaracion de variables
        // algunas inicializadas.
        int otra_var, otra_var2;
        int otra_var2=20;

        return otra_var;
}
```

Prueba 2:

```
// Prueba Incorrecta

/* Objetivo
        Comprobamos el error cometido en las
operaciones de I/O
*/

void imprime (char *msg,  f) {
        cout < s << msg << f;          // error sintactico
        cout << "\n";  // imprime un salto de línea */
        return;
}

void main (void) {
        char* s="hola Mundo!";
        int var;
        cout << s;
        cin >> s;
        cout >> s;      // error
        return;
}
```

Prueba 3:

```
// Prueba Incorrecta
// declaracion de metodo y clase
class Clase_mal {
int a,b,c;          // declaracion de metodo privado, esto esta ok!
public              // falan DOS_PUNTOS
        int metodo (); // declaracion erronea
}       // falta PUNTO_COMA
// Declaración de la clase Fecha:
class Fecha {
public: int a, b, c;
        void ponFecha (int, int, int);   /* método que recibe
                                    tres enteros y no devuelve nada */
        int daDia (void);         // método que devuelve un entero
        int daMes (void);         // método que devuelve un entero
        int daAnno (void);        // método que devuelve un entero
        void imprime (void);   /* método que no recibe
                                      ni devuelve nada */
private:
        int d,m,a;        // tres enteros privados
};
// FALTA DEFINICION DE METODOS!
// Definicion de un metodo daDia de la clase Fecha
int Fecha::daDia (void){
        return d;
}

void main (int dia) {
        Fecha var;
        return var.daDia();
}
```

Prueba 4:

```
// Prueba Incorrecta

// Declaracion de subprograma o funcion
int funcionAnodina (int n) {
        int uno = 1, cero = 0; // variables locales inicializadas

        // intrucciones operacionales
        uno = cero * "cadena";// TIPOS ERRONEOS
        uno = uno + 34 - 23 / 30 - cero * n;

        return uno;
}

// Punto ppal del programa
int main (int n) {
        int variable;
        // Asignacion de valor cierto ó falso
        variable = n * funcionAnodina(n);

//      return variable;
        return true;     // tipos de devolucion erroneo
}
```

Prueba 5:

```
// Prueba Incorrecta

// main
bool main (int n, int m)
{
	/* Declaracion de variables
	 - algunas inicializadas - */
	bool variable = true;

	// Condicional Complejo
	if variable	// FALTA CERRATURA PARENTESIS
		variable = false;
	else
		varible = true;

	// Devolvemos le valor calculado
	return variable;
}
```

Prueba 6:

```
// Prueba Incorrecta

// main
void main (void) {
	int num = 30;
	int algo =40, otro;

	if (num == 0) {
		otro -= algo;	//Operacion no contemplada
	}
	else if (algo == 0 || otro == 0) {
		otro += algo;
	}

	if (num==30) {
		num = num - num;
	} else {
		num = 30;
	} else
		num = 40;	// imposible este caso
}
```