



Campus Irapuato-Salamanca | División de Ingenierías

Universidad de Guanajuato

División de Ingenierías Campus Irapuato Salamanca

Proyecto final: Compilador

UDA: Compiladores

Impartido por: Dr. José Ruiz Pinales.

Integrantes:

José Luis Arroyo Núñez.

NUA: 390893.

Bryan Ricardo Cervantes Mancera

NUA: 146809.

### Introducción:

Como parte final en el desarrollo del compilador, se integrarán las partes anteriormente desarrolladas las cuales son el analizador léxico y sintáctico, adicional a esto se agregarán cabeceras adicionales que ayudarán a el acoplamiento y desarrollo del compilador.

### Objetivo:

Desarrollar un compilador capaz de leer instrucciones y ejecutar comandos en lenguaje C/C++, dando como resultado un análisis de código en lenguaje C y un archivo residual en lenguaje ensamblador del código que previamente se analizó.

Cabe mencionar que el compilador que se desarrollo llega hasta el punto en el que después del análisis del código fuente, este producirá código ensamblador.

### Desarrollo:

Durante el desarrollo de este proyecto hubo varios puntos clave a desarrollar en el archivo “.y” además de agregar archivos externos que ayudaron a evitar la saturación dentro del código principal, donde se definían estructuras de control da datos, funciones de definiciones de operaciones o funciones comando, a continuación, procedemos a explicar algunas de ellas:

Primeramente, debemos de contar con estructuras de datos que nos permita el almacenamiento de los datos de los tipos de variables y datos adicionales que son importantes al momento de la implementación del lenguaje de programación, en este caso el ANSI C, en este caso como se optó por la implementación de código intermedio en forma de cuádruples y triples, esta estructura se encuentra designada con el nombre de VAR\_DATA, esta estructura en conjunto con otras adicionales, las masa importantes comentadas a continuación, se encuentran definidas en la cabecera de nombre “symdefs.h”.

Las estructuras de control de símbolos o tablas de símbolos, son aquellas estructuras que nos ayudan a manejar el tipo con el cual se definen los parámetros del compilador y como estos pueden ser manejados u ocupados dentro del código para generar una cierta salida en respuesta a tipos de datos o funciones específicas, a continuación, una breve explicación de las funciones:

*/\* Definir los tipos de variables y símbolos \*/*

Esta estructura nos ayuda a definir el nombre de los símbolos, es decir ayuda a que el compilador tome las variables de referencias y las enumere secuencialmente.

```
struct symrec
{
    std::string name; /* name of symbol */
    int size;
    int init;
    int typ;
    int offset; /* data offset */
    struct {
        int func_type;
        std::list<struct symrec *> *sym_table;
    } func_desc;
    std::list<int > *dimlist;
};
```

```
typedef struct symrec symrec;
```

La siguiente enumeración nos ayuda a tener los diferentes tipos de datos que el compilador soporta, de la misma forma en esta misma tenemos declarados las funciones, estructuras de datos, uniones.

```
enum VARTYPES
{
    VOIDVAL = 1,
    CHARVAL,
    UCHARVAL,
    SCHARVAL,
    SHORTVAL,
    USHORTVAL,
    INTVAL,
    UINTVAL,
    LONGVAL,
    ULONGVAL,
    FVAL,
    DOUBLEVAL,
    LDOUBLEVAL,
    STRVAL,
    STRUCVAL,
    UNIONVAL,
    ENUMVAL,
    TYPENM
};
```

```
typedef union VALUE
{
    char charval;
    int intval;
    float floatval;
    double doubleval;
} VALUE;
```

Es una estructura que nos ayuda a tener datos punteros, para la definición de estructuras de tipo lista, como lo son los diversos ciclos: for, while, do-while, switch.

```
typedef struct VAR_DATA
{
    std::string *name;
    VALUE val;
    int init;
    int typ;
    symrec *var;
    symrec *var2;
    int plevel;
    std::list<int> *truelist;
    std::list<int> *falselist;
    std::list<VAR_DATA *> *arglist;
    std::list<int > *dimlist;
} VAR_DATA;
```

```
typedef struct CASE_DATA
{
    VALUE val;
    int typ;
    int addr;
} CASE_DATA;
```

```
typedef struct GOTO_DATA
{
    std::string *name;
    int addr;
} GOTO_DATA;
```

La estructura de combinación “unión” nos ayuda a poder definir las instrucciones que debe seguir cada regla, con ello me refiero a que definirá que tipo de tipos entran en el análisis de una regla y cuáles pueden ser admitidos como parámetros normales o como apuntadores.

```
%union
{
    struct
    {
        union
        {
            char cval;
            long int ival;
            double dval;
            char *str;
            char *name;
        };
        int type;
    } token;

    VAR_DATA *id_data; /*Datos del identificador*/
    std::list< VAR_DATA * > *idlist;
    int scsp;
    int qual;
    int typsp;
    int op;
    int typ;
    int asop;
    int instr;
    std::list<int> *sqlist;
    VAR_DATA sym; /*para poner datos sobre una variable ya declarada*/

    struct
    {
        int typq;
        int level;
    } pstruc;

    struct
    {
        std::list<int> *breaklist; //Salto a fuera de un switch
        std::list<int> *nextlist; //Para salto siguiente de la instruccion.
        std::list<int> *continuelist; //Para salto al inicio de un ciclo.
        std::list<GOTO_DATA *> *gotolist; //Para salto con GOTO
        std::list<CASE_DATA *> *caselist; //Para salto a los case de un
        switch.
    } lists;
}
```

A continuación, se presentan las definiciones de las reglas con asignación de parámetro de algún tipo, en este caso se usaron las estructuras de tipo unión para poder definir las instrucciones de las reglas, de la misma forma cada una de las definiciones cuenta con un tipo de dato especificado en la unión definida en el archivo fuente del “ansic.y”, declaración de la unión realizada anterior a esta definición:

```
%type <id_data> direct_declarator
%type <id_data> declarator
%type <id_data> init_declarator
%type <idlist> init_declarator_list
%type <scsp> storage_class_specifier
%type <sqlist> declaration_specifiers
%type <typsp> type_specifier
%type <qual> type_qualifier
%type <qual> type_qualifier_list
%type <lists> selection_statement
%type <lists> statement
%type <lists> compound_statement
%type <sym> primary_expression
%type <sym> postfix_expression
%type <sym> unary_expression
%type <sym> cast_expression
%type <sym> multiplicative_expression
%type <sym> additive_expression
%type <sym> shift_expression
%type <sym> relational_expression
%type <sym> equality_expression
%type <sym> and_expression
%type <sym> exclusive_or_expression
%type <sym> inclusive_or_expression
%type <sym> logical_and_expression
%type <sym> logical_or_expression
%type <sym> conditional_expression
%type <sym> assignment_expression
%type <sym> expression
%type <sym> argument_expression_list
%type <sym> initializer
%type <idlist> parameter_list
%type <idlist> parameter_type_list
%type <sym> parameter_declaration
%type <asop> assignment_operator
%type <lists> labeled_statement
%type <lists> iteration_statement
%type <lists> statement_list
%type <lists> jump_statement
%type <sym> expression_statement
%type <sym> constant_expression
%type <op> unary_operator
%type <typ> type_name
%type <sqlist> specifier_qualifier_list
%type <pstruc> pointer
%type <instr> N
%type <lists> M
```

```
%type <sym> logic_expression
%type <sym> logic_expression_statement
```

Consecuente a esta declaración comenzamos con la declaración de la `primary_expression`, esta es usada para darnos como resultado de la identificación de los tipos que se encontraron en las en el código fuente.

Además de esta y todo lo que conlleva intermedicamente, se declarara una regla que se denomina como `unary_expression`, esta regla es la encargada de la identificación y ejecución de expresiones como lo pueden ser, `i++`, `j--`, de la misma forma la declaración de números negativos y las negaciones.

En la regla de expresiones multiplicativas, se nos ayuda con la asignación de operadores de operaciones compuestas como lo es el casteo de datos, la multiplicación y la división, además de una de las instrucciones más importantes siendo está el módulo, pues cabe recalcar que para poder tener una declaración correcta de este operador se debe de poner doble vez en la instrucción de la regla para que el compilador lo tome como operador y no como símbolo.

```
multiplicative_expression
: cast_expression
{
    $$ = $1;
    printf("multiplicative_expression: cast_expression\n");
}
| multiplicative_expression '*' cast_expression
{
    printf("multiplicative_expression: multiplicative_expression '*'
    cast_expression\n");
    std::string *name = newtemp(); /*Crear nueva variable temporal para
    resultado*/
    symrec *s;
    s = putsym(name, gettype($1.typ, $3.typ) & (~CONST_SPEC | VAR_SPEC));
    gencode(MULT_IR, $1.var, $3.var, s);
    $$name = name;
    $$var = s;
}
| multiplicative_expression '/' cast_expression
{
    printf("multiplicative_expression: multiplicative_expression '/'
    cast_expression\n");
    std::string *name = newtemp(); /*Crear nueva variable temporal para
    resultado*/
    symrec *s;
    s = putsym(name, gettype($1.typ, $3.typ) & (~CONST_SPEC | VAR_SPEC));
    gencode(DIV_IR, $1.var, $3.var, s);
    $$name = name;
    $$var = s;
}
```

```

}
| multiplicative_expression '%' cast_expression
{
    printf("multiplicative_expression: multiplicative_expression '%%'
    cast_expression\n");
    std::string *name = newtemp(); /*Crear nueva variable temporal para
    resultado*/
    symrec *s;
    s = putsym(name, gettype($1.typ, $3.typ) & (~CONST_SPEC | VAR_SPEC));
    gencode(MOD_IR, $1.var, $3.var, s);
    $$name = name;
    $$var = s;
}
;

```

Las expresiones de adición, relación e igualdad, son expresiones que junto a las expresiones de multiplicación nos ayudan en la asignación de operadores hacia operaciones de algún tipo en el lenguaje y dentro de cualquier código para poder realizar diferentes acciones como aritméticas, lógicas o de comparación.

Una vez que se tienen las expresiones aritméticas declaradas, la parte fundamental para la comparación ya sea en la asignación o en las expresiones de control o ciclos son las expresiones lógicas las cuales las tenemos declaradas con los nombres de “and\_expression”, “exclusive\_or\_expression” e “inclusive\_or\_expression” los tres comparte similitudes de código, el cual es el siguiente:

```

printf("and_expression: and_expression '&' equality_expression\n");

std::string *name = newtemp();

symrec *s;

s=putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(AND_IR, $1.var, $3.var, s);

$$name = name;

$$var=s;

```

La única modificación que se debe de realizar en la parte del código en su segunda definición de las reglas, es en gencode donde el primer atributo cambia dependiendo de la regla en la que se aplique:

```

“and_expression”-> gencode(AND_IR, $1.var, $3.var, s);
“exclusive_or_expression”-> gencode(XOR_IR, $1.var, $3.var, s);
“inclusive_or_expression”-> gencode(OR_IR, $1.var, $3.var, s);

```



Las reglas de lógica para las instrucciones “and” y ”or” son importantes dado que para su uso se usa la regla “N” la cual retorna la dirección de la siguiente instrucción de tipo IR, es decir estas reglas nos ayudan a llevar un control y estructura de modo secuencial dentro del código estableciendo que instrucción va primero que otra y porque debe ejecutarse antes que ella.

En adición hay que recalcar que se tiene el uso de las “truelist” y “falselist” como referencia sobre cómo se ejecutara primero una que otra y en qué circunstancias es válido que se ejecute de tal forma.

```
logical_and_expression
: inclusive_or_expression
{
    $$ = $1;
    printf("logical_and_expression: inclusive_or_expression\n");
}
| logical_and_expression AND_OP N inclusive_or_expression
{
    printf("logical_and_expression: logical_and_expression AND_OP
    inclusive_or_expression\n");
    backpatch($1.truelist, $3);
    $$ .truelist = $4.truelist;
    $$ .falselist = $1.falselist;
    $$ .falselist->merge(*($4.falselist));
    $$ .typ = INTVAL;
}
;
```

```
logical_or_expression
: logical_and_expression
{
    $$ = $1;
    printf("logical_or_expression: logical_and_expression\n");
}
| logical_or_expression OR_OP N logical_and_expression
{
    printf("logical_or_expression: logical_or_expression OR_OP
    logical_and_expression\n");
    backpatch($1.falselist, $3); //El 3 es la N.
    $$ .falselist = $4.falselist;
    $$ .truelist = $1.truelist;
    $$ .truelist->merge(*($4.truelist));
    $$ .typ = INTVAL;
}
;
```

La siguiente es la regla M, la cual permite agregar la dirección de la siguiente instrucción de tipo IR en el atributo de la “nextlist” lo que genera una instrucción de salto pendiente.

```
M: /* Empty */
{
    $$nextlist = makelist(nextinstr);
    gencode(GOTO_IR);
}
```

Esta expresión es una forma alternativa, primitiva, de la declaración de una comparación if-else, que es utilizada como auxiliar en los ciclos.

conditional\_expression

```
: logical_or_expression {printf("conditional_expression: logical_or_expression\n");
```

```
$$=$1;}
```

```
| logical_or_expression '?' N expression ':'
```

```
{
    string *name = newtemp();
    symrec *s = putsym(name, $4.typ & ~CONST_SPEC | VAR_SPEC);
    gencode(STORE_IR, $4.var, NULL, s);
    $4.var = s;
}
```

```
M N conditional_expression {
```

```
printf("conditional_expression: logical_or_expression '?' expression ':' conditional_expression\n");
```

```
gencode(STORE_IR, $9.var, NULL, $4.var);
```

```
$$var = $4.var;
```

```
backpatch($1.truelist, $3);
```

```
backpatch($1.falselist, $8);
```

```
backpatch($7.nextlist, nextinstr);
```

```
$$truelist=nullptr;
```

```
$$falselist=nullptr;
```

```
}
```

```
;
```

A continuación de esto se tendrá la regla de “assignment\_expression” esta regla es la que se encarga de la asignación del tipo de expresiones que se debe de realizar, estas expresiones que se tiene contempladas son las siguientes: la operación de igualdad, multiplicación, división, modulo, operadores de bit, como lo son or, xor, not.

Con esta regla se hace énfasis en los tipos a definir de las variables, es la encargada de establecer si el dato a analizar es entero, carácter o flotante, además de que debe proveer la referencia si es que dicha variable se usa posteriormente en otras partes del código, usando como apoyo a la estructura “VAR\_SPEC” la cual provee la información necesaria para que todas las referencias a la variable inicial se traten del mismo tipo de dicha variable.

declaration

```
: declaration_specifiers ';' {printf("declaration: declaration_specifiers ';\n");}
| declaration_specifiers init_declarator_list ';'
{
    int typ;
    printf("declaration: declaration_specifiers init_declarator_list ';\n");
    typ = get_type($1);
    std::list<VAR_DATA *>::iterator it;
    cout << "Declaring Variable: " << endl;
    for(it = $2->begin(); it != $2->end(); it++)
    {
        symrec *s;
        cout << *(*it)->name << endl;
        if(isarray((*it)->typ))
        {
            installarray((*it)->name, typ | VAR_SPEC | ((*it)->typ &
~0x1F), (*it)->dimlist);
        }
        else
        {
            if(isfunction((*it)->typ))
                s = install((*it)->name, typ | FUNC_SPEC | ((*it)->typ &
~0x1F));
            else
                s = install((*it)->name, typ | VAR_SPEC | ((*it)->typ &
~0x1F));
            if((*it)->init)
            {
                if(currscope > 0) /*Verifica que sea variable local*/
                    gencode(STORE_IR, (*it)->var, nullptr, s);
            }
            else
            {
                initvar(s, (*it)->var);
            }
        }
    }
}
```

Debemos de tomar en cuenta que, como todo lenguaje, el manejo de los tipos de datos es esenciales, es por ello que debemos de especificarlos, tanto los tipos de datos, los cuales son int, char, double, float, como sus respectivos modificadores, long, short, signed, unsigned, y por ultimo las estructuras y enumeraciones que son posibles.

type\_specifier

```
: VOID {  
    printf("type_specifier: VOID\n");  
    $$ = VOID_SPEC;  
}  
| CHAR {  
    printf("type_specifier: CHAR\n");  
    $$ = CHAR_SPEC;  
}  
| SHORT {  
    printf("type_specifier: SHORT\n");  
    $$ = SHORT_SPEC;  
}  
| INT {  
    printf("type_specifier: INT\n");  
    $$ = INT_SPEC;  
}  
| LONG {  
    printf("type_specifier: LONG\n");  
    $$ = LONG_SPEC;  
}  
| FLOAT {  
    printf("type_specifier: FLOAT\n");  
    $$ = FLOAT_SPEC;  
}  
| DOUBLE {  
    printf("type_specifier: DOUBLE\n");
```

```
    $$ = DOUBLE_SPEC;
}
| SIGNED {
    printf("type_specifier: SIGNED\n");
    $$ = SIGNED_SPEC;
}
| UNSIGNED {
    printf("type_specifier: UNSIGNED\n");
    $$ = UNSIGNED_SPEC;
}
| struct_or_union_specifier {
    printf("type_specifier: struct_or_union_specifier\n");
}
| enum_specifier {
    printf("type_specifier: enum_specifier\n");
    $$ = ENUM_SPEC;
}
| TYPE_NAME {
    printf("type_specifier: TYPE_NAME\n");
    $$ = TYPENAME_SPEC;
}
;
```

La regla de declarador nos ayuda en la parte de los punteros a tener sus niveles de referencia y que tanta importancia tienen dentro del código al que estamos analizando;

```
declarator
: pointer direct_declarator
{
    printf("declarator: pointer direct_declarator\n");
    $$ = $2;
    $$->plevel = $1.level;
    $$->typ |= POINTER_SPEC | $1.typq;
}
| direct_declarator
{
    printf("declarator: direct_declarator\n");
    $$ = $1;
    if(isarray($$->typ))
    {
        //Arreglos sin dimensiones son punteros.
        if(allzero($$->dimlist))
        {
            $$->typ &= ~ARRAY_SPEC;
            $$->typ |= POINTER_SPEC;
        }
    }
    else
    {
        if(product($$->dimlist)==0) //Esta linea da error.
            yyerror("Array dimensions must be specified\n");
    }
}
;
```

La regla de declarador directo es una simple forma de controlar las referencias a donde hacen referencia los punteos, se podría describir que es un GPS de punteros, donde ayuda a que, en el análisis de código, se evite caer en una segmentación de memoria de algún tipo mientras se analiza el código, como parte de un error en la programación dentro del mismo.

Los punteros, pointer, también debemos de tomar los en cuenta en este caso, debido a que podemos tener punteros multinivel, es necesario asignar de alguna forma la especificación de esto, por lo que se optó por la implementación de esto de la siguiente forma:

pointer

```
: '*' {  
    printf("pointer: '*'\n");  
    $$typq = 0;  
    $$level = 1;  
}  
| '*' type_qualifier_list {  
    printf("pointer: '*' type_qualifier_list\n");  
    $$typq=$2;  
    $$level=1;  
}  
| '*' pointer {  
    printf("pointer: '*' pointer\n");  
    $$=$2;  
    $$level++;  
}  
| '*' type_qualifier_list pointer {  
    printf("pointer: '*' type_qualifier_list pointer\n");  
    $$=$3;  
    $$typq=$2;  
    $$level++;  
}  
;
```

El parámetro de lista y la declaración de parámetros son dos reglas que nos ayudan en la asignación de estructura de los parámetros en los punteros, es decir, ayudan a establecer el lugar de origen del puntero y hacia donde debe de llegar (apuntar), de esta manera se evita un mal uso de las direcciones de memoria dentro del código, que pueden generar un error de segmentación si es que no se tiene cuidado de a donde se apuntan las direcciones de memoria.

Para una forma de apoyo para retornar la dirección de siguiente instrucción a donde se deberá de ir:

```
N: {  
  $$=nextinstr;  
}  
;
```

La regla de “labeled\_statement” tiene como función principal ser un apoyo para el ciclo “switch” dado que delimita sus capacidades en cuanto a la estructura, principalmente en el uso de los “case” pues esta regla evita que se asignen variables incompatibles con los casos por ejemplo “case 1.5”

La regla “statement\_list” nos ayuda a vincular o unir listas, por lo cual tiene cierta relación con la regla “N” pues trabajan en conjunto para poder hacer la operación de relación entre las listas.

Esta regla nos es un auxiliar para la verificación de que las listas de verdadero o falso se encuentran vacías para de esta forma crear unas nuevas y asignarles la siguiente instrucción.

```
logic_expression : expression {  
    $$=$1;  
    if($1.truelist == nullptr || $1.falselist == nullptr)  
    {  
        std::string *name = newtemp();  
        symrec *s0;  
        s0=putsym(name, INTVAL | CONST_SPEC);  
        *(int *) (datablock+s0->offset)=0;  
    }  
}
```



```
$$truelist = makelist(nextinstr);  
gencode(IF_NE_IR, $1.var, s0, nullptr);  
$$falselist=makelist(nextinstr);  
gencode(GOTO_IR);
```

```
}
```

```
}
```

```
;
```

En esta regla tomamos en cuenta las funciones de control de flujo de datos, en los primeros lugares contamos con las dos variantes de la toma de decisión if e if-else se le agrega entre la statement la regla “N” que se está usando debido a que es en este punto es utilizado para evitar un paso adicional al evaluar la expresión determinada, en la segunda variante son agregadas de nueva cuenta la regla N como en el anterior, solo que se le adiciona después del else “M” y “N” y de nueva cuenta se realiza para evitar el paso superfluo al momento de la evaluación, y el “M” es utilizado para agregar la dirección siguiente y generar un salto pendiente ya que en caso de que el caso 1 no se cumpla tendrá que saltar de instrucción, debido a tener pendiente este salto es necesario el parchamente en cualquiera de las variante que se tenga de la instrucción if.

Por último, en esta regla tenemos definida el Switch en el cual antes de que se comience el statement es agregada la “M” por los mismos motivos mencionados anterior mente, debido a que se dará un salto y este debe de quedar de alguna forma pendiente, ya de forma interna se realizan los parchamientos correspondiente debido a este salto que se tiene pendiente.

selection\_statement

```
: IF '(' logic_expression ')' N statement %prec NO_ELSE {
    printf("selection_statement: IF '(' expression ')' statement\n");
    backpatch($3.truelist, $5);
    $$nextlist = merge($3.falselist, $6.nextlist);
    $$breaklist = $6.breaklist;
    $$continuelist = $6.continuelist;
    $$gotolist = $6.gotolist;
}
| IF '(' logic_expression ')' N statement ELSE M N statement {
    printf("selection_statement: IF '(' expression ')' statement ELSE statement\n");
    backpatch($3.truelist, $5);
    backpatch($3.falselist, $9);
    $$nextlist = merge($6.nextlist,$8.nextlist,$10.nextlist);
    $$breaklist = merge($6.breaklist,$10.breaklist);
    $$continuelist = merge($6.continuelist,$10.continuelist);
    $$gotolist = merge($6.gotolist,$10.gotolist);
}
| SWITCH '(' expression ')' M statement {
    printf("selection_statement: SWITCH '(' expression ')' statement\n");
```

```

if(!isintegral($3.typ))
{
    yyerror("Error: switch expression must be integral \n");
}

$6.breaklist=merge($6.breaklist, makelist(nextinstr));

gencode(GOTO_IR);

backpatch($5.nextlist, nextinstr);

std::list<CASE_DATA *>::iterator it;

if($6.caselist != NULL)
{
    for(it=$6.caselist->begin(); it!=$6.caselist->end(); it++)
    {
        int addr = (*it)->addr;
        int typ = (*it)->typ;
        if(typ == VOIDVAL)
        {
            gencode(GOTO_IR, addr);
        }
        else
        {
            std::string *name = newtemp();
            symrec *s = putsym(name, INTVAL | CONST_SPEC);
            symrec *s1 = $3.var;
            *(int *)((char *)datablock+s->offset) = (*it)->val.intval;

            if(isconstant($3.typ))
            {
                name=newtemp();
                s1 = putsym(name, INTVAL |CONST_SPEC);
                *(int *)((char *)datablock+s1->offset)=$3.val.intval;
            }

            gencode(IF_EQ_IR, s, s1, addr);
        }
    }
}

```

```

        }
    }
}

backpatch($6.breaklist, nextinstr);

$$breaklist = NULL;

$$continuelist = $6.continuelist;

$$gotolist = $6.gotolist;

$$nextlist = NULL;

}

;

```

La siguiente regla cuenta con la misma definición como la “logical\_expression” solamente la diferencia de esta radica en la forma en la que es empleada a posterioridad, así como también las expresiones que son utilizadas.

```

logic_expression_statement : expression_statement {
    $$=$1;

    if($1.truelist == nullptr || $1.falselist == nullptr)
    {
        std::string *name = newtemp();
        symrec *s0;

        s0 = putsym(name , INTVAL | CONST_SPEC);

        *(int *) (datablock+s0->offset)=0;

        $$truelist = makelist(nextinstr);

        gencode(IF_NE_IR, $1.var, s0, nullptr);

        $$falselist = makelist(nextinstr);

        gencode(GOTO_IR);

    }

}

;

```

La regla de “iteration\_statement” es aquella que es un auxiliar para nosotros, pues con ella podemos definir los parámetros y el comportamiento que tendrán los ciclos como “while” y “for”, además de establecer las limitaciones o restricciones que estos mismos tienen, donde definimos que es lo mínimo indispensable para que funcionen.

En esta regla hay que hacer hincapié en el uso de la regla “N” pues con ella se establecen como es que retornan las direcciones de las diferentes instrucciones de tipo IR.

Además, que, como adición, la definición de estas reglas demanda que los diferentes tipos de listas se inicialicen como nulas en ciertos puntos del ciclo para evitar que el mismo código se cicle hasta el infinito y tenga que ser forzado a detenerse, por ello tenemos los niveles de referencia que se deben seguir, esto mediante el comando “backpatch”

iteration\_statement

```
: WHILE N '(' logic_expression ')' N statement {  
    printf("iteration_statement: WHILE '(' expression ')' statement\n");  
    backpatch($7.nextlist, $2);  
    backpatch($7.continuelist, $2);  
    backpatch($4.truelist, $6);  
    $$nextlist=$4.falselist;  
    gencode(GOTO_IR, $2);  
    backpatch($7.breaklist, nextinstr);  
    $.breaklist = NULL;  
    $.continuelist = NULL;  
    $.gotolist = $7.gotolist;  
}  
| DO N statement WHILE N '(' logic_expression ')' ';' {  
    printf("iteration_statement: DO statement WHILE '(' expression ')' ';' \n");  
    backpatch($7.truelist, $2);  
    backpatch($3.nextlist, $5);  
    backpatch($3.continuelist, $5);  
    backpatch($3.breaklist, nextinstr);  
    $$nextlist=$7.falselist;  
    $.breaklist = NULL;  
    $.continuelist = NULL;
```

```

    $$gotolist = $3.gotolist;
}
| FOR '(' expression_statement N logic_expression_statement N ')' statement {
    printf("iteration_statement: FOR '(' expression_statement expression_statement ')' statement\n");
    backpatch($5.truelist, $6);
    backpatch($8.nextlist, $4);
    backpatch($8.continuelist, $4);
    $$nextlist=$5.falselist;
    gencode(GOTO_IR, $4);
    backpatch($8.breaklist, nextinstr);
    $$breaklist = NULL;
    $$continuelist = NULL;
    $$gotolist = $8.gotolist;
}
| FOR '(' expression_statement N logic_expression_statement N expression ')' M N statement {
    printf("iteration_statement: FOR '(' expression_statement expression_statement expression ')'
statement\n");
    backpatch($5.truelist, $10);
    backpatch($11.nextlist, $6);
    backpatch($11.continuelist, $6);
    backpatch($9.nextlist, $4);
    $$nextlist=$5.falselist;
    gencode(GOTO_IR, $6);
    backpatch($11.breaklist, nextinstr);
    $$breaklist = NULL;
    $$continuelist = NULL;
    $$gotolist = $11.gotolist;
}
;

```

En cuanto a la regla de “jump\_statement” su principal función está relacionada con el ciclo “switch” dado que ayuda a limitar donde inicia y termina cada caso además de que ayuda a que se limite hasta donde se usa una variable, función o apuntador dentro de cada caso del ciclo.

Sin embargo, esta regla también ayuda desde un ámbito más general, pues también ayuda a las funciones si es que requieren del uso del “return” para devolver algún tipo de dato.

jump\_statement

```
: GOTO IDENTIFIER ';' {  
    printf("jump_statement: GOTO IDENTIFIER ';' \n");  
    $$continuelist= NULL;  
    $$breaklist= NULL;  
    $$nextlist= NULL;  
    $$caselist= NULL;  
    $$gotolist= makelist(new string($2), nextinstr);  
    gencode(GOTO_IR);  
}  
| CONTINUE ';' {  
    printf("jump_statement: CONTINUE ';' \n");  
    $$continuelist= makelist(nextinstr);  
    $$breaklist= NULL;  
    $$nextlist= NULL;  
    $$caselist= NULL;  
    $$gotolist= NULL;  
    gencode(GOTO_IR);  
}  
| BREAK ';' {  
    printf("jump_statement: BREAK ';' \n");  
    $$continuelist= NULL;  
    $$breaklist= makelist(nextinstr);  
    $$nextlist= NULL;  
    $$caselist= NULL;
```

```

        $$gotolist= NULL;

        gencode(GOTO_IR);
    }
| RETURN ';' {
    printf("jump_statement: RETURN ';' \n");

    $$continuelist= NULL;

    $$breaklist= NULL;

    $$nextlist= NULL;

    $$caselist= NULL;

    $$gotolist= NULL;

    gencode(RET_IR);
}
| RETURN expression ';' {
    printf("jump_statement: RETURN expression ';' \n");

    $$continuelist= NULL;

    $$breaklist= NULL;

    $$nextlist= NULL;

    $$caselist= NULL;

    $$gotolist= NULL;

    gencode(RET_IR, $2.var, nullptr, nullptr);
}
;

```



Finalmente tenemos la última regla la cual es “function\_definition” la cual como su nombre nos indica es la manera en cómo indicamos que tipo de funciones existen dentro del código y que tipo de parámetros manejan para usar dentro de ellas, además de que también considera dentro de la función donde es que se usan los parámetros y para que si es que la función dispone de ellos o requiere de algún parámetro externo para funcionar.

```
function_definition
: declaration_specifiers declarator declaration_list compound_statement
{
    printf("function_definition : declaration_specifiers declarator
    declaration_list compound_statement\n");
    | declaration_specifiers declarator {
    int typ = get_type($1) | $2->typ;
    curr_func = install($2->name, typ);
    gencode(PROC_IR, curr_func);
    localsyms = new symboltable();
    enter_scope();
    installarg($2->arglist);
    /*Iniciar nueva tabla de etiquetas*/
    lab_table.clear();
    setargs();
    }
compound_statement
{
    printf("function_definition : declaration_specifiers declarator
    compound_statement\n");
    //printlocalvars();
    exit_scope();
    curr_func->func_desc.sym_table = localsyms;
    gencode(ENDPROC_IR, curr_func);
    //patch_gotos($4.gotolist);
    }
| declarator declaration_list compound_statement
{
    printf("function_definition : declarator declaration_list
    compound_statement\n");
    }
| declarator
{
    int typ = $1->typ;
    curr_func = install($1->name, typ);
    gencode(PROC_IR, curr_func);
    localsyms = new symboltable();
    enter_scope();
    installarg($1->arglist);
    /*Iniciar nueva tabla de etiquetas*/
    lab_table.clear();
    setargs();
    }
compound_statement
{
    printf("function_definition : declarator compound_statement\n");
```

```

        //printlocalvars();
        exit_scope();
        curr_func->func_desc.sym_table = localsyms;
        gencode(ENDPROC_IR, curr_func);
        patch_gotos($3.gotolist);
    }
;

```

Ahora como parte importante del compilador tenemos la función principal que será listada a continuación:

```

int main(int argc, char *argv[])
{
    //Guarda nombre de archivo compilarse.
    string fname;

    printf("Analizador Sintáctico de ANSI C 2021 version 0.6\n\n");

    //Crea tabla de símbolos globales
    printf("\n");
    sym_table = new symboltable();
    printf("\n");
    if(argc > 1)
    {
        yyin = fopen(argv[1], "r");
        fname = argv[1];
    }
    else
    {
        cout << "Uso: " << argv[0] << "<filename>\n";
    }

    yyparse(); //Se empieza a hacer el análisis sintáctico
    printvars(); //se termina hasta que ya no se retorna ningún token
    printf("\n");
    printf("Codigo intermedio: \n");
    print_icode();
    print_code(fname);
    return 0;
}

```

Dentro de este código lo importante es la llamada a la función “print\_code(fname)” la cual se encarga de generar un archivo ensamblador como resultado de cualquier código generado en lenguaje C y cuyo análisis haya sido exitoso.

Nota importante: el siguiente código es el que se encarga de manejar las diferentes instrucciones de tipo IR, este fragmento de código se encuentra en el archivo “genlib.h”, debido a mejoras en la optimización de código “ansic.y”

*/\* Códigos de representación intermedia \*/*

enum code\_ops

{

STORE\_IR, STOREA\_IR, LOADA\_IR, IF\_EQ\_IR, IF\_NE\_IR, IF\_LT\_IR, IF\_GT\_IR,  
IF\_LE\_IR, IF\_GE\_IR, GOTO\_IR,  
ADD\_IR, SUB\_IR, MULT\_IR, DIV\_IR,  
MINUS\_IR, MOD\_IR, INC\_IR, DEC\_IR, ADDRESS\_IR,  
DEREF\_IR, TWOCOMP\_IR, NOT\_IR, INT\_IR, FLOAT\_IR, CHAR\_IR, DOUBLE\_IR,  
LSHIFT\_IR, RSHIFT\_IR, AND\_IR, OR\_IR, XOR\_IR, RET\_IR, PROC\_IR,  
ENDPROC\_IR, CALL\_IR, PARAM\_IR

};

## Prueba del compilador:

*Nota: se omitió la parte del árbol sintáctico debido a que haría que el documento tuviera una gran extensión de hojas, por ello se adjuntan en la carpeta de compilador los códigos, capturas de pantalla y salida de la consola en archivo de texto, para que se pueda cerciorar de que en efecto ambos compiladores funcionan.*

## Prueba de compilador en MAC OS:

### Código en Lenguaje C utilizado para la prueba:

```
void printf();
void scanf();
int main()
{
    int x;
    int suma=0;
    do
    {
        printf("Ingrese un numero(si desea salir ingrese 0):\n");
        scanf("%d",&x);
        suma+=x;
    }
    while ( x != 0);
    printf("\nLa suma es: %d\n", suma);
    return 0;
}
```

### Código del árbol sintáctico:

```
Last login: Fri Jun  4 12:09:20 on ttys000
luisnunez.@MacBook-Air-de-Luis ~ % cd ejercicios
luisnunez.@MacBook-Air-de-Luis ejercicios % cd compialdor
cd: no such file or directory: compialdor
luisnunez.@MacBook-Air-de-Luis ejercicios % cd compilador
luisnunez.@MacBook-Air-de-Luis compilador % ./ccompiler do_while.c
Analizador Sintactico de ANSI C 2021 version 0.6
```

### Tabla de variables globales

Name
printf FUNC
scanf FUNC
main FUNC

### Código intermedio:

```
0: PROC main
1: suma = 0
2: PARAM @1
3: @2 = CALL printf, 1
4: @4 = &x
5: PARAM @4
6: PARAM @3
```

```

7: @5 = CALL scanf, 2
8: @6 = suma + x
9: suma = @6
10: IF x != 0 GOTO 2
11: GOTO 12
12: PARAM suma
13: PARAM @8
14: @9 = CALL printf, 2
15: RETURN 0
16: ENDPROC main

```

luisnunez.@MacBook-Air-de-Luis compilador %

Evidencia del código analizado:

```

inclusive_or_expression: exclusive_or_expression
logical_and_expression: inclusive_or_expression
logical_or_expression: logical_and_expression
conditional_expression: logical_or_expression
assignment_expression: conditional_expression
expression: assignment_expression
jump_statement: RETURN expression ';'
statement: jump_statement
statement_list: statement_list statement
compound_statement: '[' declaration_list statement_list ']'
function_definition: declaration_specifiers declarator compound_statement
external_declaration: function_definition
translation_unit: external_declaration

Tabla de variables globales
Name
printf FUNC
scanf FUNC
main FUNC

Codigo intermedio:
0: PROC main
1: suma = 0
2: PARAM @1
3: @2 = CALL printf, 1
4: @4 = &x
5: PARAM @4
6: PARAM @3
7: @5 = CALL scanf, 2
8: @6 = suma + x
9: suma = @6
10: IF x != 0 GOTO 2
11: GOTO 12
12: PARAM suma
13: PARAM @8
14: @9 = CALL printf, 2
15: RETURN 0
16: ENDPROC main

Unknown IR instruction
luisnunez.@MacBook-Air-de-Luis compilador %

```

Código en Lenguaje C utilizado para la prueba:

```
void printf();
void scanf();
int main()
{
    int n,i,facto=1;
    printf("Programa que hace el factorial de un numero n\n");
    printf("Ingrese un numero n:\n");
    scanf("%i",&n);

    for(i=1;i<=n;i++)
    {
        facto*=i;
    }
    printf("El factorial de %i es: %i\n\n",n,facto);

    return 0;
}
```

Código del árbol sintáctico:

Tabla de variables globales

Name

printf FUNC

scanf FUNC

main FUNC

Código intermedio:

```
0: PROC main
1: facto = 1
2: PARAM @1
3: @2 = CALL printf, 1
4: PARAM @3
5: @4 = CALL printf, 1
6: @6 = &n
7: PARAM @6
8: PARAM @5
9: @7 = CALL scanf, 2
10: i = 1
11: IF i <= n GOTO 16
12: GOTO 19
13: @9 = i
14: i = i + 1
15: GOTO 11
16: @10 = facto * i
17: facto = @10
18: GOTO 13
19: PARAM facto
20: PARAM n
21: PARAM @11
```

22: @12 = CALL printf, 3  
23: RETURN 0  
24: ENDPROC main

## Evidencia del código analizado:

```
statement_list: statement_list statement
compound_statement: '[' declaration_list statement_list ']'
function_definition: declaration_specifiers declarator compound_statement
external_declaration: function_definition
translation_unit: external_declaration

Tabla de variables globales
Name
printf FUNC
scanf FUNC
main FUNC

Codigo intermedio:
0: PROC main
1: fact0 = 1
2: PARAM @1
3: @2 = CALL printf, 1
4: PARAM @3
5: @4 = CALL printf, 1
6: @6 = &n
7: PARAM @6
8: PARAM @5
9: @7 = CALL scanf, 2
10: i = 1
11: IF i <= n GOTO 16
12: GOTO 19
13: @9 = i
14: i = i + 1
15: GOTO 11
16: @10 = fact0 * i
17: fact0 = @10
18: GOTO 13
19: PARAM fact0
20: PARAM n
21: PARAM @11
22: @12 = CALL printf, 3
23: RETURN 0
24: ENDPROC main

Unknown IR instruction
luisnunez. @MacBook-Air-de-Luis compilador % |
```

## Prueba de compilador en Linux Ubuntu:

### Anidación de ciclos For, uso de While, uso de Do-While y Switch

#### Código usado para la prueba:

```
Void printf();
void scanf();

int main()
{
    char opcion="a,b,c";

    inicio:
    printf("----- \n");
    printf("Elija una opcion: \n");
    printf("a)promedio con ciclo while \n");
    printf("b)promedio con ciclo do-while \n");
    printf("c)promedio con ciclo for \n");
    printf("\n");
    /*__fpurge(stdin);*/
    scanf("%c",&opcion);

    switch(opcion)
    {
    case 'a' :
    {
        int nd;
        int i=0;
        float dato, prom, contador=0;

        printf("Numero de datos a ingresar: \n");
        scanf("%i",&nd);
        while(i < nd)
        {
            i++;
            printf("Digite el dato numero %i: \n",i);
            scanf("%f",&dato);

            if(dato < 0)
            {
                break;
            }

            contador += dato;
        }

        if (nd == i)
        {
            prom=contador/nd;
            printf("Promedio %.3f \n",prom);
        }
        else
        {
            prom=contador/(i-1);
            printf("Promedio %.3f \n",prom);
        }
        break;
    }
    case 'b':
    {
        int nd, i=0, dato, j=0;
        float prom, contador=0;

        printf("Numero de datos a ingresar: \n");
        scanf("%i",&nd);

        do
        {
            i++;
            j++;
```



```

printf("Digite el dato numero %i: \n",i);
scanf("%i",&dato);
if(dato < 0)
{
break;
}
contador += dato;

/*i++;
//j++;*/
}
while(i < nd);

if (nd == j && nd == 0)
{
prom=contador/nd;
printf("Promedio %.3f \n",prom);
}
else
{
prom=contador/(j-1);
printf("Promedio %.3f \n",prom);
}
break;
}
case 'c':
{
int nd, i, j=0;
float dato, prom, contador=0;

printf("Numero de datos a ingresar: \n");
scanf("%i",&nd);

for(i=1; i<=nd; i++)
{
j++;
printf("Digite en dato numero %i: \n",i);
scanf("%f",&dato);
contador += dato;

if(dato < 0)
{
break;
}
}

if (nd == j)
{
prom=contador/nd;
printf("Promedio %.3f \n",prom);
}
else
{
prom=contador/(j-1);
printf("Promedio %.3f \n",prom);
}
break;
}

default:
{
printf("No corresponde a una opcion \n");
goto inicio;
}
break;
}

return 0;
}

```

**Código resultante de analizacion:**

Tabla de variables globales

Name  
printf FUNC  
scanf FUNC  
main FUNC

Codigo intermedio:

```
0: PROC main
1: opcion = @1
2: PARAM @3
3: @4 = CALL printf, 1
4: PARAM @6
5: @7 = CALL printf, 1
6: PARAM @9
7: @10 = CALL printf, 1
8: PARAM @12
9: @13 = CALL printf, 1
10: PARAM @15
11: @16 = CALL printf, 1
12: PARAM @18
13: @19 = CALL printf, 1
14: @22 = &opcion
15: PARAM @22
16: PARAM @21
17: @23 = CALL scanf, 2
18: GOTO 152
19: i = 0
20: contador = 0
21: PARAM @28
22: @29 = CALL printf, 1
23: @32 = &nd
24: PARAM @32
25: PARAM @31
26: @33 = CALL scanf, 2
27: IF i < nd GOTO 29
28: GOTO 44
29: @34 = i
30: i = i + 1
31: PARAM i
32: PARAM @36
33: @37 = CALL printf, 2
34: @40 = &dato
35: PARAM @40
36: PARAM @39
37: @41 = CALL scanf, 2
38: IF dato < 0 GOTO 40
39: GOTO 41
40: GOTO 44
41: @43 = contador + dato
42: contador = @43
43: GOTO 27
44: IF nd == i GOTO 46
45: GOTO 52
46: @44 = contador / nd
47: prom = @44
48: PARAM prom
49: PARAM @46
50: @47 = CALL printf, 2
51: GOTO 58
52: @49 = i - 1
53: @50 = contador / @49
54: prom = @50
55: PARAM prom
56: PARAM @52
57: @53 = CALL printf, 2
58: GOTO 156
59: i = 0
60: j = 0
61: contador = 0
62: PARAM @59
63: @60 = CALL printf, 1
64: @63 = &nd
65: PARAM @63
66: PARAM @62
```

```
67: @64 = CALL scanf, 2
68: @65 = i
69: i = i + 1
70: @66 = j
71: j = j + 1
72: PARAM i
73: PARAM @68
74: @69 = CALL printf, 2
75: @72 = &dato
76: PARAM @72
77: PARAM @71
78: @73 = CALL scanf, 2
79: IF dato < 0 GOTO 81
80: GOTO 82
81: GOTO 86
82: @75 = contador + dato
83: contador = @75
84: IF i < nd GOTO 68
85: GOTO 86
86: IF nd == j GOTO 88
87: GOTO 96
88: IF nd == 0 GOTO 90
89: GOTO 96
90: @77 = contador / nd
91: prom = @77
92: PARAM prom
93: PARAM @79
94: @80 = CALL printf, 2
95: GOTO 102
96: @82 = j - 1
97: @83 = contador / @82
98: prom = @83
99: PARAM prom
100: PARAM @85
101: @86 = CALL printf, 2
102: GOTO 156
103: j = 0
104: contador = 0
105: PARAM @91
106: @92 = CALL printf, 1
107: @95 = &nd
108: PARAM @95
109: PARAM @94
110: @96 = CALL scanf, 2
111: i = 1
112: IF i <= nd GOTO 117
113: GOTO 132
114: @98 = i
115: i = i + 1
116: GOTO 112
117: @99 = j
118: j = j + 1
119: PARAM i
120: PARAM @101
121: @102 = CALL printf, 2
122: @105 = &dato
123: PARAM @105
124: PARAM @104
125: @106 = CALL scanf, 2
126: @107 = contador + dato
127: contador = @107
128: IF dato < 0 GOTO 130
129: GOTO 114
130: GOTO 132
131: GOTO 114
132: IF nd == j GOTO 134
133: GOTO 140
134: @109 = contador / nd
135: prom = @109
136: PARAM prom
137: PARAM @111
138: @112 = CALL printf, 2
139: GOTO 146
140: @114 = j - 1
141: @115 = contador / @114
```

Evidencia de código analizado:

```

1 0000 74
2 0001 30
3 0002 30
4 0003 + constant / den
5 0004 + constant / den
6 0005 + den
7 0006 1
8 0007 0001 + 1 0000 40
9 0008 1
10 0009 + constant / rd
11 0010 0
12 0011 0000
13 0012 0000
14 0013 + 0000
15 0014 + 0000
16 0015 0000
17 0016 0000
18 0017 + 0000
19 0018 0000
20 0019 + 0000
21 0020 0000
22 0021 0000
23 0022 + 0000
24 0023 0000
25 0024 + 0000
26 0025 0000
27 0026 + 0000
28 0027 0000
29 0028 + 0000
30 0029 0000
31 0030 + 0000
32 0031 + 0000
33 0032 + 0000
34 0033 + 0000
35 0034 + 0000
36 0035 + 0000
37 0036 + 0000
38 0037 + 0000
39 0038 + 0000
40 0039 + 0000
41 0040 + 0000
42 0041 + 0000
43 0042 + 0000
44 0043 + 0000
45 0044 + 0000
46 0045 + 0000
47 0046 + 0000
48 0047 + 0000
49 0048 + 0000
50 0049 + 0000
51 0050 + 0000
52 0051 + 0000
53 0052 + 0000
54 0053 + 0000
55 0054 + 0000
56 0055 + 0000
57 0056 + 0000
58 0057 + 0000
59 0058 + 0000
60 0059 + 0000
61 0060 + 0000
62 0061 + 0000
63 0062 + 0000
64 0063 + 0000
65 0064 + 0000
66 0065 + 0000
67 0066 + 0000
68 0067 + 0000
69 0068 + 0000
70 0069 + 0000
71 0070 + 0000
72 0071 + 0000
73 0072 + 0000
74 0073 + 0000
75 0074 + 0000
76 0075 + 0000
77 0076 + 0000
78 0077 + 0000
79 0078 + 0000
80 0079 + 0000
81 0080 + 0000
82 0081 + 0000
83 0082 + 0000
84 0083 + 0000
85 0084 + 0000
86 0085 + 0000
87 0086 + 0000
88 0087 + 0000
89 0088 + 0000
90 0089 + 0000
91 0090 + 0000
92 0091 + 0000
93 0092 + 0000
94 0093 + 0000
95 0094 + 0000
96 0095 + 0000
97 0096 + 0000
98 0097 + 0000
99 0098 + 0000
100 0099 + 0000
101 0100 + 0000
102 0101 + 0000
103 0102 + 0000
104 0103 + 0000
105 0104 + 0000
106 0105 + 0000
107 0106 + 0000
108 0107 + 0000
109 0108 + 0000
110 0109 + 0000
111 0110 + 0000
112 0111 + 0000
113 0112 + 0000
114 0113 + 0000
115 0114 + 0000
116 0115 + 0000
117 0116 + 0000
118 0117 + 0000
119 0118 + 0000
120 0119 + 0000
121 0120 + 0000
122 0121 + 0000
123 0122 + 0000
124 0123 + 0000
125 0124 + 0000
126 0125 + 0000
127 0126 + 0000
128 0127 + 0000
129 0128 + 0000
130 0129 + 0000
131 0130 + 0000
132 0131 + 0000
133 0132 + 0000
134 0133 + 0000
135 0134 + 0000
136 0135 + 0000
137 0136 + 0000
138 0137 + 0000
139 0138 + 0000
140 0139 + 0000
141 0140 + 0000
142 0141 + 0000
143 0142 + 0000
144 0143 + 0000
145 0144 + 0000
146 0145 + 0000
147 0146 + 0000
148 0147 + 0000
149 0148 + 0000
150 0149 + 0000
151 0150 + 0000
152 0151 + 0000
153 0152 + 0000
154 0153 + 0000
155 0154 + 0000
156 0155 + 0000
157 0156 + 0000
158 0157 + 0000
159 0158 + 0000
160 0159 + 0000
161 0160 + 0000
162 0161 + 0000
163 0162 + 0000
164 0163 + 0000
165 0164 + 0000
166 0165 + 0000
167 0166 + 0000
168 0167 + 0000
169 0168 + 0000
170 0169 + 0000
171 0170 + 0000
172 0171 + 0000
173 0172 + 0000
174 0173 + 0000
175 0174 + 0000
176 0175 + 0000
177 0176 + 0000
178 0177 + 0000
179 0178 + 0000
180 0179 + 0000
181 0180 + 0000
182 0181 + 0000
183 0182 + 0000
184 0183 + 0000
185 0184 + 0000
186 0185 + 0000
187 0186 + 0000
188 0187 + 0000
189 0188 + 0000
190 0189 + 0000
191 0190 + 0000
192 0191 + 0000
193 0192 + 0000
194 0193 + 0000
195 0194 + 0000
196 0195 + 0000
197 0196 + 0000
198 0197 + 0000
199 0198 + 0000
200 0199 + 0000
201 0200 + 0000
202 0201 + 0000
203 0202 + 0000
204 0203 + 0000
205 0204 + 0000
206 0205 + 0000
207 0206 + 0000
208 0207 + 0000
209 0208 + 0000
210 0209 + 0000
211 0210 + 0000
212 0211 + 0000
213 0212 + 0000
214 0213 + 0000
215 0214 + 0000
216 0215 + 0000
217 0216 + 0000
218 0217 + 0000
219 0218 + 0000
220 0219 + 0000
221 0220 + 0000
222 0221 + 0000
223 0222 + 0000
224 0223 + 0000
225 0224 + 0000
226 0225 + 0000
227 0226 + 0000
228 0227 + 0000
229 0228 + 0000
230 0229 + 0000
231 0230 + 0000
232 0231 + 0000
233 0232 + 0000
234 0233 + 0000
235 0234 + 0000
236 0235 + 0000
237 0236 + 0000
238 0237 + 0000
239 0238 + 0000
240 0239 + 0000
241 0240 + 0000
242 0241 + 0000
243 0242 + 0000
244 0243 + 0000
245 0244 + 0000
246 0245 + 0000
247 0246 + 0000
248 0247 + 0000
249 0248 + 0000
250 0249 + 0000
251 0250 + 0000
252 0251 + 0000
253 0252 + 0000
254 0253 + 0000
255 0254 + 0000
256 0255 + 0000
257 0256 + 0000
258 0257 + 0000
259 0258 + 0000
260 0259 + 0000
261 0260 + 0000
262 0261 + 0000
263 0262 + 0000
264 0263 + 0000
265 0264 + 0000
266 0265 + 0000
267 0266 + 0000
268 0267 + 0000
269 0268 + 0000
270 0269 + 0000
271 0270 + 0000
272 0271 + 0000
273 0272 + 0000
274 0273 + 0000
275 0274 + 0000
276 0275 + 0000
277 0276 + 0000
278 0277 + 0000
279 0278 + 0000
280 0279 + 0000
281 0280 + 0000
282 0281 + 0000
283
```

## Anidación de ciclos For y Switch

### Código usado para la prueba:

```
int i, nl, max, min, hl, ne, j, l;
int cap[28];
char ans= "S,s";

void printf();
void scanf();

int main()
{
    char op="a,b,c";
intento:
    nl=0;
    printf("Numero de datos a capturar menor a 30 \n");
    scanf("%i",&nl);
    if(nl >=30)
    {
        printf("\n");
        printf("Numero de datos a capturar excede el limite \n");
        printf("Vuelve a intentar \n");
        goto intento;
    }

    for(l=0; l<nl; l++)
    {
        printf("Dato %i \n", l+1);
        scanf("%i",&cap[l]);
    }

    menu:
    printf("\n_____ \n");
    printf("Que desea realizar con los datos ingresados: \n");
    printf("a) Buscar cual dato ingresado es el mayor \n");
    printf("b) Buscar cual dato ingresado es el menor \n");
    printf("c) Ordenar los Datos ingresados \n");
    scanf("%s",&op);

    switch(op)
    {
        case 'a':
        {
            printf("Buscar cual dato ingresado es el mayor \n");
            max=cap[0];
            for(i=0; i<nl; i++)
            {
                if(cap[i]>max)
                {
                    max=cap[i];
                }
            }
            printf("El valor mayor ingresado es el %i",max);
            printf("\n_____ \n");
            printf("¿Desea seleccionar otra opcion del menu?\n");
            printf("Si su respuesta es si ingrese S/s, de lo contrario ingrese cualquier otro caracter \n");
            scanf("%s",&ans);
            if(ans == 'S' || ans == 's')
            {
                goto menu;
            }
            break;
        }

        case 'b':
        {
            printf("Buscar cual dato ingresado es el menor \n");
            min=cap[0];
            for(i=0; i<nl; i++)
            {
```

```

if(cap[i]<min)
{
min=cap[i];
}
}
printf("El valor menor ingresado es el %i",min);
printf("\n_____ \n");
printf("¿Desea seleccionar otra opcion del menu?\n");
printf("Si su respuesta es si ingrese S/s, de lo contrario ingrese cualquier otro caracter \n");
scanf("%s",&ans);
if(ans == 'S' || ans =='s')
{
goto menu;
}
break;
}

case 'c':
{
printf("Ordenar los Datos ingresados \n");
for(j=0; j<nl; j++)
{
for(i=j+1; i<nl; i++)
if(cap[j] > cap[i])
{
hl=cap[j];
cap[j] = cap[i];
cap[i]=hl;
}
}

printf("Datos ordenados en orden ascendente \n");
printf("\n");
for(i=0; i<nl; i++)
{
printf("\t %i\n",cap[i]);
}

for(j=0; j<nl; j++)
{
for(i=j+1; i<nl; i++)
if(cap[j] < cap[i])
{
hl=cap[j];
cap[j] = cap[i];
cap[i]=hl;
}
}

printf("\n");
printf("Datos ordenados en orden ascendente \n");
for(i=0; i<nl; i++)
{
printf("\t %i\n",cap[i]);
}

printf("\n_____ \n");
printf("¿Desea seleccionar otra opcion del menu?\n");
printf("Si su respuesta es si ingrese S/s, de lo contrario ingrese cualquier otro caracter \n");
scanf("%s",&ans);
if(ans == 'S' || ans =='s')
{
goto menu;
}
break;
}
default:
{
printf("Opcion incorrecta intente de nuevo... \n");
goto menu;
}
}

return 0;

```

}

Codigo resultante de analizacion:

Tabla de variables globales

Name

i VAR

nl VAR

max VAR

min VAR

hl VAR

ne VAR

j VAR

l VAR

@0 CONST

cap VAR

@2 CONST

ans VAR

printf FUNC

scanf FUNC

main FUNC

Codigo intermedio:

0: PROC main

1: op = @4

2: nl = 0

3: PARAM @7

4: @8 = CALL printf, 1

5: @11 = &nl

6: PARAM @11

7: PARAM @10

8: @12 = CALL scanf, 2

9: IF nl >= 30 GOTO 11

10: GOTO 18

11: PARAM @15

12: @16 = CALL printf, 1

13: PARAM @18

14: @19 = CALL printf, 1

15: PARAM @21

16: @22 = CALL printf, 1

17: GOTO 2

18: l = 0

19: IF l < nl GOTO 24

20: GOTO 35

21: @24 = l

22: l = l + 1

23: GOTO 19

24: @28 = l + 1

25: PARAM @28

26: PARAM @26

27: @29 = CALL printf, 2

28: @33 = l \* @32

29: @34 = cap[@33]

30: @35 = &@34

```
31: PARAM @35
32: PARAM @31
33: @36 = CALL scanf, 2
34: GOTO 21
35: PARAM @38
36: @39 = CALL printf, 1
37: PARAM @41
38: @42 = CALL printf, 1
39: PARAM @44
40: @45 = CALL printf, 1
41: PARAM @47
42: @48 = CALL printf, 1
43: PARAM @50
44: @51 = CALL printf, 1
45: @54 = &op
46: PARAM @54
47: PARAM @53
48: @55 = CALL scanf, 2
49: GOTO 240
50: PARAM @58
51: @59 = CALL printf, 1
52: @62 = 0 * @61
53: @63 = cap[@62]
54: max = @63
55: i = 0
56: IF i < nl GOTO 61
57: GOTO 69
58: @65 = i
59: i = i + 1
60: GOTO 56
61: @67 = i * @66
62: @68 = cap[@67]
63: IF @68 > max GOTO 65
64: GOTO 58
65: @70 = i * @69
66: @71 = cap[@70]
67: max = @71
68: GOTO 58
69: PARAM max
70: PARAM @73
71: @74 = CALL printf, 2
72: PARAM @76
73: @77 = CALL printf, 1
74: PARAM @79
75: @80 = CALL printf, 1
76: PARAM @82
77: @83 = CALL printf, 1
78: @86 = &ans
79: PARAM @86
80: PARAM @85
81: @87 = CALL scanf, 2
82: IF ans == @88 GOTO 86
83: GOTO 84
84: IF ans == @89 GOTO 86
```



```
85: GOTO 87
86: GOTO 35
87: GOTO 244
88: PARAM @92
89: @93 = CALL printf, 1
90: @96 = 0 * @95
91: @97 = cap[@96]
92: min = @97
93: i = 0
94: IF i < nl GOTO 99
95: GOTO 107
96: @99 = i
97: i = i + 1
98: GOTO 94
99: @101 = i * @100
100: @102 = cap[@101]
101: IF @102 < min GOTO 103
102: GOTO 96
103: @104 = i * @103
104: @105 = cap[@104]
105: min = @105
106: GOTO 96
107: PARAM min
108: PARAM @107
109: @108 = CALL printf, 2
110: PARAM @110
111: @111 = CALL printf, 1
112: PARAM @113
113: @114 = CALL printf, 1
114: PARAM @116
115: @117 = CALL printf, 1
116: @120 = &ans
117: PARAM @120
118: PARAM @119
119: @121 = CALL scanf, 2
120: IF ans == @122 GOTO 124
121: GOTO 122
122: IF ans == @123 GOTO 124
123: GOTO 125
124: GOTO 35
125: GOTO 244
126: PARAM @126
127: @127 = CALL printf, 1
128: j = 0
129: IF j < nl GOTO 134
130: GOTO 158
131: @129 = j
132: j = j + 1
133: GOTO 129
134: @131 = j + 1
135: i = @131
136: IF i < nl GOTO 141
137: GOTO 131
138: @132 = i
```

```
139: i = i + 1
140: GOTO 136
141: @134 = j * @133
142: @135 = cap[@134]
143: @137 = i * @136
144: @138 = cap[@137]
145: IF @135 > @138 GOTO 147
146: GOTO 138
147: @140 = j * @139
148: @141 = cap[@140]
149: hl = @141
150: @143 = j * @142
151: @145 = i * @144
152: @146 = cap[@145]
153: cap[@143] = @146
154: @148 = i * @147
155: cap[@148] = hl
156: GOTO 138
157: GOTO 131
158: PARAM @150
159: @151 = CALL printf, 1
160: PARAM @153
161: @154 = CALL printf, 1
162: i = 0
163: IF i < nl GOTO 168
164: GOTO 174
165: @156 = i
166: i = i + 1
167: GOTO 163
168: @160 = i * @159
169: @161 = cap[@160]
170: PARAM @161
171: PARAM @158
172: @162 = CALL printf, 2
173: GOTO 165
174: j = 0
175: IF j < nl GOTO 180
176: GOTO 204
177: @164 = j
178: j = j + 1
179: GOTO 175
180: @166 = j + 1
181: i = @166
182: IF i < nl GOTO 187
183: GOTO 177
184: @167 = i
185: i = i + 1
186: GOTO 182
187: @169 = j * @168
188: @170 = cap[@169]
189: @172 = i * @171
190: @173 = cap[@172]
191: IF @170 < @173 GOTO 193
192: GOTO 184
```

```
193: @175 = j * @174
194: @176 = cap[@175]
195: hl = @176
196: @178 = j * @177
197: @180 = i * @179
198: @181 = cap[@180]
199: cap[@178] = @181
200: @183 = i * @182
201: cap[@183] = hl
202: GOTO 184
203: GOTO 177
204: PARAM @185
205: @186 = CALL printf, 1
206: PARAM @188
207: @189 = CALL printf, 1
208: i = 0
209: IF i < nl GOTO 214
210: GOTO 220
211: @191 = i
212: i = i + 1
213: GOTO 209
214: @195 = i * @194
215: @196 = cap[@195]
216: PARAM @196
217: PARAM @193
218: @197 = CALL printf, 2
219: GOTO 211
220: PARAM @199
221: @200 = CALL printf, 1
222: PARAM @202
223: @203 = CALL printf, 1
224: PARAM @205
225: @206 = CALL printf, 1
226: @209 = &ans
227: PARAM @209
228: PARAM @208
229: @210 = CALL scanf, 2
230: IF ans == @211 GOTO 234
231: GOTO 232
232: IF ans == @212 GOTO 234
233: GOTO 235
234: GOTO 35
235: GOTO 244
236: PARAM @214
237: @215 = CALL printf, 1
238: GOTO 35
239: GOTO 244
240: IF 97 == op GOTO 50
241: IF 98 == op GOTO 88
242: IF 99 == op GOTO 126
243: GOTO 236
244: RETURN 0
245: ENDPROC main
```

Evidencia de código analizado:

```

100: GOTO 136
101: B114 = j - #113
102: B115 = com[B114]
103: B117 = i - #116
104: B118 = com[B117]
105: IF B115 = B118 GOTO 147
106: GOTO 136
107: B140 = i - #119
108: B141 = com[B140]
109: B1 = B111
110: B143 = j - #142
111: B145 = i - #144
112: B146 = com[B141]
113: com[B145] = B146
114: B148 = i - #147
115: com[B148] = B1
116: GOTO 136
117: GOTO 135
118: PAVAN B100
119: B113 = CAL printf, 1
120: PAVAN B101
121: B116 = CAL printf, 1
122: l = #
123: IF l = #1 GOTO 168
124: GOTO 174
125: B150 = l
126: IF B150 = l
127: GOTO 168
128: PAVAN B102
129: B151 = com[B150]
130: B1 = PAVAN B100
131: B152 = CAL printf, 2
132: GOTO 168
133: l = #
134: IF j = #1 GOTO 168
135: GOTO 206
136: B154 = j
137: B155 = j - #1
138: GOTO 175
139: B156 = j - #1
140: l = B156
141: GOTO 167
142: IF l = #1 GOTO 167
143: B157 = l
144: B1 = l - #1
145: GOTO 167
146: B159 = j - #168
147: B170 = com[B159]
148: B172 = i - #171
149: B173 = com[B172]
150: IF B170 = B173 GOTO 193
151: GOTO 166
152: B175 = j - #174
153: B176 = com[B175]
154: B1 = B170
155: B178 = j - #177
156: B180 = i - #179
157: B181 = com[B180]
158: com[B178] = B181
159: B183 = i - #182
160: com[B183] = B1
161: GOTO 166
162: GOTO 164
163: GOTO 177
164: PAVAN B103
165: B186 = CAL printf, 1
166: PAVAN B104
167: B189 = CAL printf, 1
168: l = #
169: IF l = #1 GOTO 214
170: GOTO 228
171: B191 = l
172: l = #1
173: IF GOTO 206
174: B195 = i - #194
175: B196 = com[B195]
176: PAVAN B105
177: PAVAN B106
178: B197 = CAL printf, 2
179: GOTO 211
180: PAVAN B107
181: B199 = CAL printf, 1
182: PAVAN B108
183: B201 = CAL printf, 1
184: B202 = CAL printf, 1
185: B209 = B199
186: PAVAN B109
187: B210 = CAL printf, 1
188: B218 = CAL scanf, 2
189: IF B199 = B210 GOTO 234
190: GOTO 212
191: GOTO 215
192: IF B199 = B210 GOTO 234
193: GOTO 215
194: GOTO 244
195: GOTO 215
196: PAVAN B114
197: B213 = CAL printf, 1
198: GOTO 215
199: GOTO 244
200: IF j = #1 GOTO 248
201: IF j = #2 GOTO 248
202: IF j = #3 GOTO 250
203: GOTO 236
204: B250 = j
205: B250C = B250
206: B250C = B250C * 10
207: B250C = B250C + B250
208: B250C = B250C * 10
209: B250C = B250C + B250
210: B250C = B250C * 10
211: B250C = B250C + B250
212: B250C = B250C * 10
213: B250C = B250C + B250
214: B250C = B250C * 10
215: B250C = B250C + B250
216: B250C = B250C * 10
217: B250C = B250C + B250
218: B250C = B250C * 10
219: B250C = B250C + B250
220: B250C = B250C * 10
221: B250C = B250C + B250
222: B250C = B250C * 10
223: B250C = B250C + B250
224: B250C = B250C * 10
225: B250C = B250C + B250
226: B250C = B250C * 10
227: B250C = B250C + B250
228: B250C = B250C * 10
229: B250C = B250C + B250
230: B250C = B250C * 10
231: B250C = B250C + B250
232: B250C = B250C * 10
233: B250C = B250C + B250
234: B250C = B250C * 10
235: B250C = B250C + B250
236: B250C = B250C * 10
237: B250C = B250C + B250
238: B250C = B250C * 10
239: B250C = B250C + B250
240: B250C = B250C * 10
241: B250C = B250C + B250
242: B250C = B250C * 10
243: B250C = B250C + B250
244: B250C = B250C * 10
245: B250C = B250C + B250
246: B250C = B250C * 10
247: B250C = B250C + B250
248: B250C = B250C * 10
249: B250C = B250C + B250
250: B250C = B250C * 10
251: B250C = B250C + B250
252: B250C = B250C * 10
253: B250C = B250C + B250
254: B250C = B250C * 10
255: B250C = B250C + B250
256: B250C = B250C * 10
257: B250C = B250C + B250
258: B250C = B250C * 10
259: B250C = B250C + B250
260: B250C = B250C * 10
261: B250C = B250C + B250
262: B250C = B250C * 10
263: B250C = B250C + B250
264: B250C = B250C * 10
265: B250C = B250C + B250
266: B250C = B250C * 10
267: B250C = B250C + B250
268: B250C = B250C * 10
269: B250C = B250C + B250
270: B250C = B250C * 10
271: B250C = B250C + B250
272: B250C = B250C * 10
273: B250C = B250C + B250
274: B250C = B250C * 10
275: B250C = B250C + B250
276: B250C = B250C * 10
277: B250C = B250C + B250
278: B250C = B250C * 10
279: B250C = B250C + B250
280: B250C = B250C * 10
281: B250C = B250C + B250
282: B250C = B250C * 10
283: B250C = B250C + B250
284: B250C = B250C * 10
285: B250C = B250C + B250
286: B250C = B250C * 10
287: B250C = B250C + B250
288: B250C = B250C * 10
289: B250C = B250C + B250
290: B250C = B250C * 10
291: B250C = B250C + B250
292: B250C = B250C * 10
293: B250C = B250C + B250
294: B250C = B250C * 10
295: B250C = B250C + B250
296: B250C = B250C * 10
297: B250C = B250C + B250
298: B250C = B250C * 10
299: B250C = B250C + B250
300: B250C = B250C * 10
301: B250C = B250C + B250
302: B250C = B250C * 10
303: B250C = B250C + B250
304: B250C = B250C * 10
305: B250C = B250C + B250
306: B250C = B250C * 10
307: B250C = B250C + B250
308: B250C = B250C * 10
309: B250C = B250C + B250
310: B250C = B250C * 10
311: B250C = B250C + B250
312: B250C = B250C * 10
313: B250C = B250C + B250
314: B250C = B250C * 10
315: B250C = B250C + B250
316: B250C = B250C * 10
317: B250C = B250C + B250
318: B250C = B250C * 10
319: B250C = B250C + B250
320: B250C = B250C * 10
321: B250C = B250C + B250
322: B250C = B250C * 10
323: B250C = B250C + B250
324: B250C = B250C * 10
325: B250C = B250C + B250
326: B250C = B250C * 10
327: B250C = B250C + B250
328: B250C = B250C * 10
329: B250C = B250C + B250
330: B250C = B250C * 10
331: B250C = B250C + B250
332: B250C = B250C * 10
333: B250C = B250C + B250
334: B250C = B250C * 10
335: B25
```

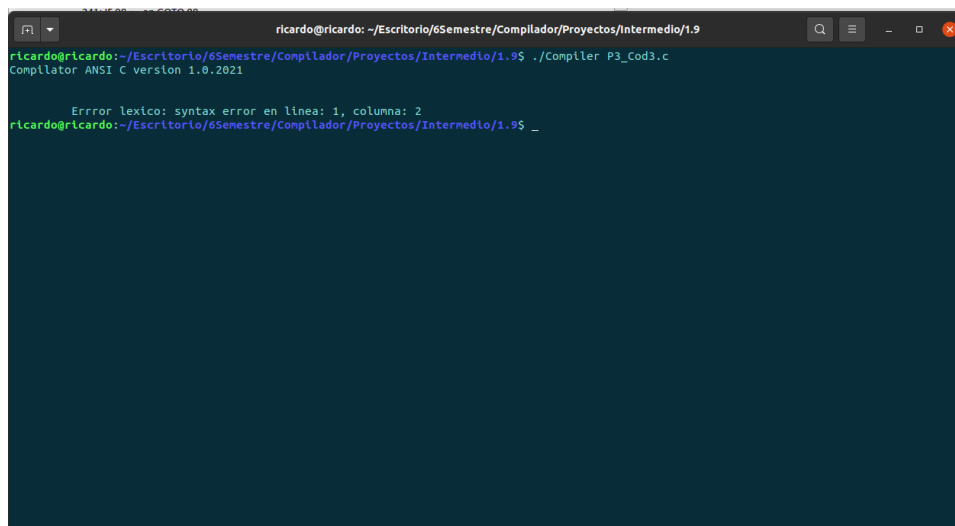
Nota: los códigos .asm estarán en las carpetas compiladas de los compiladores, para evitar saturación de código en el documento y sea posible un análisis eficiente.

## Conclusiones:

Fue un proyecto bastante laborioso de hacer principalmente por la parte de segmentar el código en diferentes archivos, en total 5 (ansic.l, ansic.y, genlib.h, symdef.h, symfunc.h), lo que dificultó un tanto que el proyecto funcionara correctamente en las primeras versiones de compilación, pero pudimos resolver estos problemas para al final tener operando al 100% nuestros compiladores, además de que entretenido como distintas reglas iban relacionándose entre sí y como todo el conjunto de dichas reglas daba como resultado final la salida en consola un árbol sintáctico producto del análisis de código además de una tabla de símbolos y un análisis en código intermedio en formato ensamblador.

Una observación a realizar con respecto a el funcionamiento del compilador final, el cual al momento de tener comentarios que tienen como antecesor “//Comentario” marca error en esa línea y solamente deja que sea posible los comentarios con el formato “/\*comentario\*/”

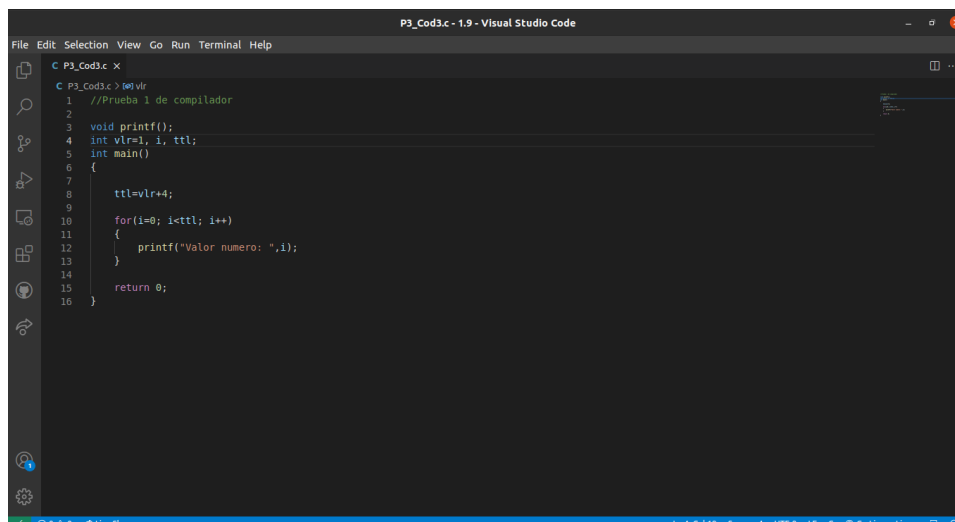
## Evidencia:



```
ricardo@ricardo: ~/Escritorio/6Semestre/Compilador/Proyectos/Intermedio/1.9
ricardo@ricardo:~/Escritorio/6Semestre/Compilador/Proyectos/Intermedio/1.9$ ./Compiler P3_Cod3.c
Compiler ANSI C version 1.0.2021

Error lexico: syntax error en línea: 1, columna: 2
ricardo@ricardo:~/Escritorio/6Semestre/Compilador/Proyectos/Intermedio/1.9$ _
```

## Código prueba:



```
P3_Cod3.c - 1.9 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
C P3_Cod3.c x
C P3_Cod3.c > |@ vir
1 //Prueba 1 de compilador
2
3 void printf();
4 int vlr=1, i, ttl;
5 int main()
6 {
7     ttl=vlr+4;
8
9     for(i=0; i<ttl; i++)
10     {
11         printf("Valor numero: ",i);
12     }
13
14     return 0;
15 }
16
```