Lenguaje Micro

INTRODUCCIÓN AL PROCESO DE COMPILACIÓN

El programa que se debe compilar es una secuencia de caracteres que termina con un centinela.

CONCEPTOS BÁSICOS

El proceso de compilación está formado por dos partes:

1. el ANÁLISIS y
2. la SÍNTESIS.

EL ANÁLISIS DEL PROGRAMA FUENTE

En la compilación, el análisis está formado por tres fases:

a) el Análisis Léxico,
b) el Análisis Sintáctico, y
c) el Análisis Semántico.

ANÁLISIS LÉXICO

Detecta LEXEMAS y los catregoriza en CATEGORÍAS LÉXICAS o TOKENS

ANÁLISIS SINTÁCTICO

- ✔ tendrá la capacidad de determinar si las construcciones que componen el programa son sintácticamente correctas.
- ✔ Sin embargo, no podrá determinar si el programa, en su totalidad, es sintácticamente correcto.

ANÁLISIS SEMÁNTICO

El Análisis Semántico complementa lo que hizo el Análisis Sintáctico.

Las rutinas semánticas llevan a cabo dos funciones:

1. Chequean la *semántica estática*.
2. Si la construcción es semánticamente correcta, hacen la *traducción*;

La Sintaxis se divide, normalmente, en

1. componentes Independientes del Contexto y
2. componentes Sensibles al Contexto.

El componente semántico de un LP se divide, habitualmente, en dos clases:

1. *Semántica Estática*, y
2. *Semántica en Tiempo de Ejecución*.

UN COMPILADOR SIMPLE

El único tipo de dato es entero.
Todos los identificadores son declarados implícitamente y con una longitud máxima de 32 caracteres.
Los identificadores deben comenzar con una letra y están compuestos de letras y dígitos. Las constantes son secuencias de dígitos (números enteros). - Hay dos tipos de sentencias:
Asignación ID := Expresión;
expresión es infija y se construye con identificadores, constantes y los operadores + y –; los paréntesis están permitidos. J
Entrada/Salida leer (lista de IDs); escribir (lista de Expresiones);
Cada sentencia termina con un "punto y coma" (;).
El cuerpo de un programa está delimitado por inicio y fin.
inicio, **fin**, **leer** y **escribir** son palabras reservadas y deben escribirse en minúscula.

Gramática Léxica

<token> -> *uno de* <identificador> <constante> <palabraReservada>
<operadorAditivo> <asignación> <carácterPuntuación>
<identificador> -> <letra> {<letra o dígito>}
<constante> -> <dígito> {dígito>}
<letra o dígito> -> *uno de* <letra> <dígito>
<letra> -> *una de* **a-z A-Z**
<dígito> -> *uno de* **0-9**
<palabraReservada> -> *una de* **inicio fin leer escribir**
<operadorAditivo> -> *uno de* **+ -**
<asignación> -> **:=**
<carácterPuntuación> -> *uno de* **( ) , ;**

Gramática Sintáctica

<objetivo> -> <programa> fdt
<programa> -> **inicio** <listaSentencias> **fin**
<listaSentencias> -> <sentencia> {<sentencia>}
<sentencia> -> <identificador> **:=** <expresión> **;** |
leer (<listaIdentificadores>) **;** |
escribir (<listaExpresiones>) **;**
<listaIdentificadores> -> <identificador> {**,** <identificador>}
<listaExpresiones> -> <expresión> {**,** <expresión>}
<expresión> -> <primaria> {<operadorAditivo> <primaria>}
<primaria> -> <identificador> | <constante> | **(** <expresión> **)**

Al construir el compilador, cada tokens tendrá su propio nombre

En el Programa Fuente	Nombre del Token
Inicio	INICIO
Fin	FIN
Leer	LEER
Escribir	ESCRIBIR
:=	ASIGNACIÓN
(	PARENIZQUIERDO
)	PARENDERECHO
,	COMA
;	PUNTOYCOMA
+	SUMA
-	RESTA

Aproximacion a C

```
/* Compilador del Lenguaje Micro (Fischer)        */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define NUMESTADOS 15
#define NUMCOLS 13
#define TAMLEX 32+1
#define TAMNOM 20+1
/*****************Declaraciones Globales*************************/
FILE * in;
typedef enum
{
 INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO, PARENDERECHO, PUNTOYCOMA,
COMA, ASIGNACION, SUMA, RESTA, FDT, ERRORLEXICO
} TOKEN;
```

El11ÁLISIS LÉXICO Scanner. Lee uno a uno, los caracteres que forman un lexema y produce representaciones de tokens.
El ANÁLISIS SINTÁCTICO Parser procesa los tokens invoca a la rutina semántica
Existen dos formas de Análisis Sintáctico:

- ✔ el Análisis Sintáctico Descendente, que permite ser construido por un programador
- ✔ el Análisis Sintáctico Ascendente (conocido como *bottom-up*), que requiere el auxilio de un programa especializado tipo *yacc*.

ÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO (ASDR).

- ✔ utiliza rutinas, que pueden ser recursivas

- ✔ va “construyendo” un árbol de análisis sintáctico (AAS).

Árbol de Análisis Sintáctico

parte del axioma de una GIC y tiene las siguientes propiedades:

1. La raíz está etiquetada con el axioma de la GIC.
2. Cada hoja está etiquetada con un token. Si se leen de izquierda a derecha, las hojas representan la construcción derivada.
3. Cada nodo interior está etiquetado con un noterminal.

La TABLA DE SÍMBOLOS (TS) es una estructura de datos compleja que es utilizada para el almacenamiento de todos los identificadores del programa a compilar.
Cada elemento de la TS está formado por una cadena y sus atributos. En el caso de Micro, la TS contendrá las palabras reservadas y los identificadores; cada entrada tendrá, como único atributo, un código que indique si la cadena representa una “palabra reservada” o “un identificador”.
En general, la TS es muy utilizada durante toda la compilación y, específicamente, en la etapa de Análisis por las rutinas semánticas.

Aproximacion a C

```
typedef struct
{
 char identifi[TAMLEX];
 TOKEN t;          /* t=0, 1, 2, 3 Palabra Reservada, t=ID=4 Identificador */
} RegTS;
RegTS TS[1000] = { {"inicio", INICIO}, {"fin", FIN}, {"leer", LEER}, {"escribir", ESCRIBIR}, {"$", 99} };

typedef struct
{
 TOKEN clase;
 char nombre[TAMLEX];
 int valor;
} REG_EXPRESION;

char buffer[TAMLEX];
TOKEN tokenActual;
int flagToken = 0;
```

En el ASDR cada noterminal tiene asociado un PAS
Cada PAS sigue el desarrollo del lado derecho de la producción que implementa

1) Si se debe procesar un noterminal <A>, se invoca al PAS correspondiente.
2) Para procesar un terminal t, se invoca al procedimiento Match con argumento t.

```
/************************Scanner**********************************/
TOKEN scanner()
{
 int tabla[NUMESTADOS][NUMCOLS] = { {  1,  3,  5,  6,  7,  8,  9, 10, 11, 14, 13,  0, 14 },
                                    {  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    {  4,  3,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 12, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 } };
 int car;
 int col;
 int estado = 0;
 int i = 0;
  do
 {
  car = fgetc(in);
  col = columna(car);
  estado = tabla[estado][col];
  if ( col != 11 )
  {
   buffer[i] = car;
   i++;
  }
 }
 while ( !estadoFinal(estado) && !(estado == 14) );
 buffer[i] = '\0';

 switch ( estado )
 {
  case 2 : if ( col != 11 )
               {
                ungetc(car, in);
                buffer[i-1] = '\0';
               }
               return ID;
  case 4 : if ( col != 11 )
               {
                ungetc(car, in);
                buffer[i-1] = '\0';
               }
         return CONSTANTE;
  case 5 : return SUMA;
```

```
 case 6 : return RESTA;
 case 7 : return PARENIZQUIERDO;
 case 8 : return PARENDERECHO;
 case 9 : return COMA;
 case 10 : return PUNTOYCOMA;
 case 12 : return ASIGNACION;
 case 13 : return FDT;
 case 14 : return ERRORLEXICO;
 }
 return 0;
}
int estadoFinal(int e)
{
 if ( e == 0 || e == 1 || e == 3 || e == 11 || e == 14 ) return 0;
 return 1;
}
int columna(int c)
{
 if ( isalpha(c) ) return 0;
 if ( isdigit(c) ) return 1;
 if ( c == '+' ) return 2;
 if ( c == '-' ) return 3;
 if ( c == '(' ) return 4;
 if ( c == ')' ) return 5;
 if ( c == ',' ) return 6;
 if ( c == ';' ) return 7;
 if ( c == ':' ) return 8;
 if ( c == '=' ) return 9;
 if ( c == EOF ) return 10;
 if ( isspace(c) ) return 11;
 return 12;
}
/*************Fin Scanner*********************************************/
```

```
Aproximacion a C
/**********Procedimientos de Analisis Sintactico (PAS) ****************/
void Objetivo(void)
{
 /* <objetivo> -> <programa> FDT #terminar */
 Programa();
 Match(FDT);
 Terminar();
}
void Programa(void)
{
 /* <programa> -> #comenzar INICIO <listaSentencias> FIN */
 Comenzar();
 Match(INICIO);
 ListaSentencias();
 Match(FIN);
}
void ListaSentencias(void)
{
 /* <listaSentencias> -> <sentencia> {<sentencia>} */
 Sentencia();
 while ( 1 )
 {
  switch ( ProximoToken() )
  {
   case ID : case LEER : case ESCRIBIR :
   Sentencia();
   break;
   default : return;
  }
 }
}

void Sentencia(void)
{
 TOKEN tok = ProximoToken();
 REG_EXPRESION izq, der;
 switch ( tok )
 {
  case ID :          /* <sentencia> -> ID := <expresion> #asignar ; */
   Identificador(&izq);
   Match(ASIGNACION);
   Expresion(&der);
   Asignar(izq, der);
   Match(PUNTOYCOMA);
   break;
  case LEER :       /* <sentencia> -> LEER ( <listaIdentificadores> ) */
   Match(LEER);
   Match(PARENIZQUIERDO);
   ListaIdentificadores();
   Match(PARENDERECHO);
```

```
  Match(PUNTOYCOMA);
  break;
 case ESCRIBIR :            /* <sentencia> -> ESCRIBIR ( <listaExpresiones> ) */
  Match(ESCRIBIR);
  Match(PARENIZQUIERDO);
  ListaExpresiones();
  Match(PARENDERECHO);
  Match(PUNTOYCOMA);
  break;
 default : return;
 }
}

void ListaIdentificadores(void)
{
 /* <listaIdentificadores> -> <identificador> #leer_id {COMA <identificador> #leer_id} */
 TOKEN t;
 REG_EXPRESION reg;
 Identificador(&reg);
 Leer(reg);
 for ( t = ProximoToken(); t == COMA; t = ProximoToken() )
 {
  Match(COMA);
  Identificador(&reg);
  Leer(reg);
 }
}

void Identificador(REG_EXPRESION * presul)
{
 /* <identificador> -> ID #procesar_id */
 Match(ID);
 *presul = ProcesarId();
}

void ListaExpresiones(void)
{
 /* <listaExpresiones> -> <expresion> #escribir_exp {COMA <expresion> #escribir_exp} */

 TOKEN t;
 REG_EXPRESION reg;
 Expresion(&reg);
 Escribir(reg);
 for ( t = ProximoToken(); t == COMA; t = ProximoToken() )
 {
  Match(COMA);
  Expresion(&reg);
  Escribir(reg);
 }
}
```

```
void Expresion(REG_EXPRESION * presul)
{
 /* <expresion> -> <primaria> { <operadorAditivo> <primaria> #gen_infijo } */
 REG_EXPRESION operandoIzq, operandoDer;
 char op[TAMLEX];
 TOKEN t;
 Primaria(&operandoIzq);
 for ( t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken() )
 {
  OperadorAditivo(op);
  Primaria(&operandoDer);
  operandoIzq = GenInfijo(operandoIzq, op, operandoDer);
 }
 *presul = operandoIzq;
}

void Primaria(REG_EXPRESION * presul)
{
 TOKEN tok = ProximoToken();
 switch ( tok )
 {
  case ID :          /* <primaria> -> <identificador> */
   Identificador(presul);
   break;
  case CONSTANTE :       /* <primaria> -> CONSTANTE #procesar_cte */
   Match(CONSTANTE);
   *presul = ProcesarCte();
   break;
  case PARENIZQUIERDO :          /* <primaria> -> PARENIZQUIERDO <expresion> PARENDERECHO */
   Match(PARENIZQUIERDO);
   Expresion(presul);
   Match(PARENDERECHO);
   break;
  default : return;
 }
}

void OperadorAditivo(char * presul)
{
 /* <operadorAditivo> -> SUMA #procesar_op | RESTA #procesar_op */
 TOKEN t = ProximoToken();
 if ( t == SUMA || t == RESTA )
 {
  Match(t);
  strcpy(presul, ProcesarOp());
 }
 else
  ErrorSintactico(t);
}
```

```
/*********************Rutinas Semanticas*****************************/
REG_EXPRESION ProcesarCte(void)
{
 /* Convierte cadena que representa numero a numero entero y construye un registro semantico */
 REG_EXPRESION reg;
 reg.clase = CONSTANTE;
 strcpy(reg.nombre, buffer);
 sscanf(buffer, "%d", &reg.valor);
 return reg;
}

REG_EXPRESION ProcesarId(void)
{
 /* Declara ID y construye el correspondiente registro semantico */
 REG_EXPRESION reg;
 Chequear(buffer);
 reg.clase = ID;
 strcpy(reg.nombre, buffer);
 return reg;
}

char * ProcesarOp(void)
{
 /* Declara OP y construye el correspondiente registro semantico */
 return buffer;
}

void Leer(REG_EXPRESION in)
{
 /* Genera la instruccion para leer */
 Generar("Read", in.nombre, "Entera", "");
}

void Escribir(REG_EXPRESION out)
{
 /* Genera la instruccion para escribir */
 Generar("Write", Extraer(&out), "Entera", "");
}

REG_EXPRESION GenInfijo(REG_EXPRESION e1, char * op, REG_EXPRESION e2)
{
 /* Genera la instruccion para una operacion infija y construye un registro semantico con el resultado
*/
 REG_EXPRESION reg;
 static unsigned int numTemp = 1;
 char cadTemp[TAMLEX] ="Temp&";
 char cadNum[TAMLEX];
 char cadOp[TAMLEX];
 if ( op[0] == '-' ) strcpy(cadOp, "Restar");
 if ( op[0] == '+' ) strcpy(cadOp, "Sumar");
```

```
 sprintf(cadNum, "%d", numTemp);
 numTemp++;
 strcat(cadTemp, cadNum);
 if ( e1.clase == ID) Chequear(Extraer(&e1));
 if ( e2.clase == ID) Chequear(Extraer(&e2));
 Chequear(cadTemp);
 Generar(cadOp, Extraer(&e1), Extraer(&e2), cadTemp);
 strcpy(reg.nombre, cadTemp);
 return reg;
}
/***************Funciones Auxiliares*********************************/
void Match(TOKEN t)
{
 if ( !(t == ProximoToken()) ) ErrorSintactico();
 flagToken = 0;
}

TOKEN ProximoToken()
{
 if ( !flagToken )
 {
  tokenActual = scanner();
  if ( tokenActual == ERRORLEXICO ) ErrorLexico();
  flagToken = 1;
  if ( tokenActual == ID )
  {
   Buscar(buffer, TS, &tokenActual);
  }
 }
 return tokenActual;
}
void ErrorLexico()
{
 printf("Error Lexico\n");
}
void ErrorSintactico()
{
 printf("Error Sintactico\n");
}
void Generar(char * co, char * a, char * b, char * c)
{
 /* Produce la salida de la instruccion para la MV por stdout */
 printf("%s %s%c%s%c%s\n", co, a, ',', b, ',', c);
}
char * Extraer(REG_EXPRESION * preg)
{
 /* Retorna la cadena del registro semantico */
 return preg->nombre;
}
int Buscar(char * id, RegTS * TS, TOKEN * t)
{
```

```
/* Determina si un identificador esta en la TS */

int i = 0;
while ( strcmp("$", TS[i].identifi) )
{
 if ( !strcmp(id, TS[i].identifi) )
 {
  *t = TS[i].t;
  return 1;
 }
 i++;
}
return 0;
}
void Colocar(char * id, RegTS * TS)
{
/* Agrega un identificador a la TS */
int i = 4;
while ( strcmp("$", TS[i].identifi) ) i++;
 if ( i < 999 )
{
 strcpy(TS[i].identifi, id );
 TS[i].t = ID;
 strcpy(TS[++i].identifi, "$" );
}
}
void Chequear(char * s)
{
/* Si la cadena No esta en la Tabla de Simbolos la agrega,
   y si es el nombre de una variable genera la instruccion */
TOKEN t;
if ( !Buscar(s, TS, &t) )
{
 Colocar(s, TS);
 Generar("Declara", s, "Entera", "");
}
}
void Comenzar(void)
{
/* Inicializaciones Semanticas */
}
void Terminar(void)
{
/* Genera la instruccion para terminar la ejecucion del programa */
Generar("Detiene", "", "", "");
}
void Asignar(REG_EXPRESION izq, REG_EXPRESION der)
{
/* Genera la instruccion para la asignacion */
Generar("Almacena", Extraer(&der), izq.nombre, "");
}
```

TIPOS DE ANÁLISIS SINTÁCTICOS Y DE GICs

Al derivar una secuencia de tokens, si existe más de un noterminal en una cadena de derivación debemos elegir cuál es el próximo noterminal que se va a expandir. Por ello se utilizan dos tipos de derivaciones:

- ✔ DERIVACIÓN POR IZQUIERDA (o *Derivación a Izquierda*): ocurre cuando siempre se reemplaza el primer noterminal que se encuentre en una cadena de derivación leída de izquierda a derecha;
- ✔ DERIVACIÓN POR DERECHA (o *Derivación a Derecha*): ocurre cuando siempre reemplaza el último noterminal de la cadena de derivación leída de izquierda a derecha.

El Análisis Sintáctico Descendente (*top-down*, en inglés) produce una Derivación por Izquierda
La Derivación por Derecha se utiliza en el Análisis Sintáctico Ascendente (*bottom-up*, en inglés),

Un Parser Ascendente utiliza una Derivación a Derecha, pero en orden inverso, esto es: la última producción aplicada en la Derivación a Derecha, es la primera produción que es"descubierta", mientras que la primera producción utilizada, la que involucra al axioma, es la última producción en ser "descubierta". En otras palabras, "reduce el árbol de análisis sintáctico" hasta llegar al axioma. En conclusión: la secuencia de producciones reconocida en el Análisis Sintáctico Ascendente es exactamente la inversa a la secuencia de producciones que forma la Derivación a Derecha.

Sea la GIC con producciones:

```
1 S -> aST
2 S -> b
3 T -> cT
4 T -> d
```

Supongamos que debemos analizar si **`aabcdd`** es sintácticamente correcta.

1º Derivación a Izquierda – Análisis Sintáctico Descendente (comienza en el axioma)
se reemplaza el primer noterminal que se encuentre en una cadena de derivación leída de izquierda a derecha

Cadena de Derivación Obtenida	Próxima Producción a Aplicar
`S (axioma)`	`S -> aST (1)`
`aST`	`S -> aST (1)`
`aaSTT`	`S -> b (2)`
`aabTT`	`T -> cT (3)`
`aabcTT`	`T -> d (4)`
`aabcdT`	`T -> d (4)`
`aabcdd correcta`	

2º Derivación a Derecha
reemplaza el último noterminal de la cadena de derivación leída de izquierda a derecha.

Cadena de Derivación Obtenida	Próxima Producción a Aplicar
`S (axioma)`	`S -> aST (1)`
`aST`	`T -> d (4)`
`aSd`	`S -> aST (1)`
`aaSTd`	`T -> cT (3)`
`aaScTd`	`T -> d (4)`
`aaScdd`	`S -> b (2)`

```
aabcdd
```

3º Análisis Sintáctico Ascendente (a partir de la tabla anterior, en orden inverso)
Derivación a Derecha, pero en orden inverso, esto es: la última producción aplicada en la Derivación a Derecha, es la primera produción que es "descubierta", mientras que la primera es la última producción en ser "descubierta".

Cadena de Derivación	Próxima Producción a Aplicar
aabcdd	S -> b (2)
aaScdd	T -> d (4)
aaScTd	T -> cT (3)
aaSTd	S -> aST (1)
aSd	T -> d (4)
aST	S -> aST (1)
S correcta	

/* Compilador del Lenguaje Micro (Fischer) */

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define NUMESTADOS 15
#define NUMCOLS 13
#define TAMLEX 32+1
#define TAMNOM 20+1
/******************Declaraciones Globales************************/
FILE * in;
typedef enum
{
 INICIO, FIN, LEER, ESCRIBIR, ID, CONSTANTE, PARENIZQUIERDO, PARENDERECHO, PUNTOYCOMA,
COMA, ASIGNACION, SUMA, RESTA, FDT, ERRORLEXICO
} TOKEN;
typedef struct
{
 char identifi[TAMLEX];
 TOKEN t;          /* t=0, 1, 2, 3 Palabra Reservada, t=ID=4 Identificador */
} RegTS;
RegTS TS[1000] = { {"inicio", INICIO}, {"fin", FIN}, {"leer", LEER}, {"escribir", ESCRIBIR}, {"$", 99} };

typedef struct
{
 TOKEN clase;
 char nombre[TAMLEX];
 int valor;
} REG_EXPRESION;

char buffer[TAMLEX];
TOKEN tokenActual;
int flagToken = 0;

/**********************Prototipos de Funciones************************/
TOKEN scanner();
int columna(int c);
int estadoFinal(int e);
void Objetivo(void);
void Programa(void);
void ListaSentencias(void);
void Sentencia(void);
void ListaIdentificadores(void);
void Identificador(REG_EXPRESION * presul);
void ListaExpresiones(void);
void Expresion(REG_EXPRESION * presul);
void Primaria(REG_EXPRESION * presul);
void OperadorAditivo(char * presul);

REG_EXPRESION ProcesarCte(void);
```

```
REG_EXPRESION ProcesarId(void);
char * ProcesarOp(void);
void Leer(REG_EXPRESION in);
void Escribir(REG_EXPRESION out);
REG_EXPRESION GenInfijo(REG_EXPRESION e1, char * op, REG_EXPRESION e2);

void Match(TOKEN t);
TOKEN ProximoToken();
void ErrorLexico();
void ErrorSintactico();
void Generar(char * co, char * a, char * b, char * c);
char * Extraer(REG_EXPRESION * preg);
int Buscar(char * id, RegTS * TS, TOKEN * t);
void Colocar(char * id, RegTS * TS);
void Chequear(char * s);
void Comenzar(void);
void Terminar(void);
void Asignar(REG_EXPRESION izq, REG_EXPRESION der);

/**************************Programa Principal************************/

int main(int argc, char * argv[])
{
 TOKEN tok;
 char nomArchi[TAMNOM];
 int l;

/**************************Se abre el Archivo Fuente*****************/
 if ( argc == 1 )
 {
  printf("Debe ingresar el nombre del archivo fuente (en lenguaje Micro) en la linea de
comandos\n");
  return -1;
 }
 if ( argc != 2 )
 {
  printf("Numero incorrecto de argumentos\n");
  return -1;
 }
 strcpy(nomArchi, argv[1]);
 l = strlen(nomArchi);
 if ( l > TAMNOM )
 {
  printf("Nombre incorrecto del Archivo Fuente\n");
  return -1;
 }
 if ( nomArchi[l-1] != 'm' || nomArchi[l-2] != '.' )
 {
  printf("Nombre incorrecto del Archivo Fuente\n");
  return -1;
 }
```

```
 if ( (in = fopen(nomArchi, "r") ) == NULL )
 {
  printf("No se pudo abrir archivo fuente\n");
  return -1;
 }

/************************Inicio Compilacion**************************/

 Objetivo();

/*************************Se cierra el Archivo Fuente*****************/

 fclose(in);

 return 0;
}

/**********Procedimientos de Analisis Sintactico (PAS) ****************/

void Objetivo(void)
{
 /* <objetivo> -> <programa> FDT #terminar */

 Programa();
 Match(FDT);
 Terminar();
}

void Programa(void)
{
 /* <programa> -> #comenzar INICIO <listaSentencias> FIN */

 Comenzar();// invocacion a las rutinas semanticas, en la gramatica se coloca con #
 Match(INICIO);
 ListaSentencias();

 Match(FIN);
}

void ListaSentencias(void)
{
 /* <listaSentencias> -> <sentencia> {<sentencia>} */

 Sentencia();

 while ( 1 )
 {
  switch ( ProximoToken() )
  {
   case ID : case LEER : case ESCRIBIR :
```

```
  Sentencia();
  break;
  default : return;
 }
 }
}

void Sentencia(void)
{
 TOKEN tok = ProximoToken();
 REG_EXPRESION izq, der;

 switch ( tok )
 {
  case ID :          /* <sentencia> -> ID := <expresion> #asignar ; (rutina semantica)*/
   Identificador(&izq);
   Match(ASIGNACION);
   Expresion(&der);
   Asignar(izq, der);
   Match(PUNTOYCOMA);
   break;
  case LEER :      /* <sentencia> -> LEER ( <listaIdentificadores> ) */
   Match(LEER);
   Match(PARENIZQUIERDO);
   ListaIdentificadores();
   Match(PARENDERECHO);
   Match(PUNTOYCOMA);
   break;
  case ESCRIBIR :            /* <sentencia> -> ESCRIBIR ( <listaExpresiones> ) */
   Match(ESCRIBIR);
   Match(PARENIZQUIERDO);
   ListaExpresiones();
   Match(PARENDERECHO);
   Match(PUNTOYCOMA);
   break;
  default : return;
 }
}

void ListaIdentificadores(void)
{
 /* <listaIdentificadores> -> <identificador> #leer_id {COMA <identificador> #leer_id} */

 TOKEN t;
 REG_EXPRESION reg;

 Identificador(&reg);
 Leer(reg);

 for ( t = ProximoToken(); t == COMA; t = ProximoToken() )
 {
```

```
 Match(COMA);
 Identificador(&reg);
 Leer(reg);
 }
}

void Identificador(REG_EXPRESION * presul)
{
 /* <identificador> -> ID #procesar_id */

 Match(ID);
 *presul = ProcesarId();
}

void ListaExpresiones(void)
{
 /* <listaExpresiones> -> <expresion> #escribir_exp {COMA <expresion> #escribir_exp} */

 TOKEN t;
 REG_EXPRESION reg;

 Expresion(&reg);
 Escribir(reg);

 for ( t = ProximoToken(); t == COMA; t = ProximoToken() )
 {
  Match(COMA);
  Expresion(&reg);
  Escribir(reg);
 }
}

void Expresion(REG_EXPRESION * presul)
{
 /* <expresion> -> <primaria> { <operadorAditivo> <primaria> #gen_infijo } */

 REG_EXPRESION operandoIzq, operandoDer;
 char op[TAMLEX];
 TOKEN t;

 Primaria(&operandoIzq);

 for ( t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken() )
 {
  OperadorAditivo(op);
  Primaria(&operandoDer);
  operandoIzq = GenInfijo(operandoIzq, op, operandoDer);
 }
 *presul = operandoIzq;
}
```

```
void Primaria(REG_EXPRESION * presul)
{
 TOKEN tok = ProximoToken();

 switch ( tok )
 {
  case ID :          /* <primaria> -> <identificador> */
   Identificador(presul);
   break;
  case CONSTANTE :       /* <primaria> -> CONSTANTE #procesar_cte */
   Match(CONSTANTE);
   *presul = ProcesarCte();
   break;
  case PARENIZQUIERDO :          /* <primaria> -> PARENIZQUIERDO <expresion> PARENDERECHO */
   Match(PARENIZQUIERDO);
   Expresion(presul);
   Match(PARENDERECHO);
   break;
  default : return;
 }
}

void OperadorAditivo(char * presul)
{
 /* <operadorAditivo> -> SUMA #procesar_op | RESTA #procesar_op */

 TOKEN t = ProximoToken();

 if ( t == SUMA || t == RESTA )
 {
  Match(t);
  strcpy(presul, ProcesarOp());
 }
 else
  ErrorSintactico(t);
}

/**********************Rutinas Semanticas*****************************/

REG_EXPRESION ProcesarCte(void)
{
 /* Convierte cadena que representa numero a numero entero y construye un registro semantico */

 REG_EXPRESION reg;

 reg.clase = CONSTANTE;
 strcpy(reg.nombre, buffer);
 sscanf(buffer, "%d", &reg.valor);

 return reg;
}
```

```
REG_EXPRESION ProcesarId(void)
{
 /* Declara ID y construye el correspondiente registro semantico */

 REG_EXPRESION reg;

 Chequear(buffer);
 reg.clase = ID;
 strcpy(reg.nombre, buffer);

 return reg;
}

char * ProcesarOp(void)
{
 /* Declara OP y construye el correspondiente registro semantico */

 return buffer;
}

void Leer(REG_EXPRESION in)
{
 /* Genera la instruccion para leer */

 Generar("Read", in.nombre, "Entera", "");
}

void Escribir(REG_EXPRESION out)
{
 /* Genera la instruccion para escribir */

 Generar("Write", Extraer(&out), "Entera", "");
}

REG_EXPRESION GenInfijo(REG_EXPRESION e1, char * op, REG_EXPRESION e2)
{
 /* Genera la instruccion para una operacion infija y construye un registro semantico con el resultado
*/

 REG_EXPRESION reg;
 static unsigned int numTemp = 1;
 char cadTemp[TAMLEX] ="Temp&";
 char cadNum[TAMLEX];
 char cadOp[TAMLEX];

 if ( op[0] == '-' ) strcpy(cadOp, "Restar");
 if ( op[0] == '+' ) strcpy(cadOp, "Sumar");

 sprintf(cadNum, "%d", numTemp);
 numTemp++;
```

```
 strcat(cadTemp, cadNum);

 if ( e1.clase == ID) Chequear(Extraer(&e1));
 if ( e2.clase == ID) Chequear(Extraer(&e2));
 Chequear(cadTemp);
 Generar(cadOp, Extraer(&e1), Extraer(&e2), cadTemp);

 strcpy(reg.nombre, cadTemp);

 return reg;
}
/***************Funciones Auxiliares**********************************/

void Match(TOKEN t)
{
 if ( !(t == ProximoToken()) ) ErrorSintactico();
 flagToken = 0;
}

TOKEN ProximoToken()
{
 if ( !flagToken )
 {
  tokenActual = scanner();
  if ( tokenActual == ERRORLEXICO ) ErrorLexico();
  flagToken = 1;
  if ( tokenActual == ID )
  {
   Buscar(buffer, TS, &tokenActual);
  }
 }

 return tokenActual;
}

void ErrorLexico()
{
 printf("Error Lexico\n");
}

void ErrorSintactico()
{
 printf("Error Sintactico\n");
}

void Generar(char * co, char * a, char * b, char * c)
{
 /* Produce la salida de la instruccion para la MV por stdout */

 printf("%s %s%c%s%c%s\n", co, a, ',', b, ',', c);
}
```

```
char * Extraer(REG_EXPRESION * preg)
{
 /* Retorna la cadena del registro semantico */

 return preg->nombre;
}

int Buscar(char * id, RegTS * TS, TOKEN * t)
{
 /* Determina si un identificador esta en la TS */

 int i = 0;

 while ( strcmp("$", TS[i].identifi) )
 {
  if ( !strcmp(id, TS[i].identifi) )
  {
   *t = TS[i].t;
   return 1;
  }
  i++;
 }
 return 0;
}

void Colocar(char * id, RegTS * TS)
{
 /* Agrega un identificador a la TS */

 int i = 4;

 while ( strcmp("$", TS[i].identifi) ) i++;

 if ( i < 999 )
 {
  strcpy(TS[i].identifi, id );
  TS[i].t = ID;
  strcpy(TS[++i].identifi, "$" );
 }
}

void Chequear(char * s)
{
 /* Si la cadena No esta en la Tabla de Simbolos la agrega,
    y si es el nombre de una variable genera la instruccion */

 TOKEN t;

 if ( !Buscar(s, TS, &t) )
 {
```

```
  Colocar(s, TS);
  Generar("Declara", s, "Entera", "");
 }
}

void Comenzar(void)
{
 /* Inicializaciones Semanticas */
}

void Terminar(void)
{
 /* Genera la instruccion para terminar la ejecucion del programa */

 Generar("Detiene", "", "", "");
}

void Asignar(REG_EXPRESION izq, REG_EXPRESION der)
{
 /* Genera la instruccion para la asignacion */

 Generar("Almacena", Extraer(&der), izq.nombre, "");
}

/************************Scanner************************************/

TOKEN scanner()
{
 int tabla[NUMESTADOS][NUMCOLS] = { {  1,  3,  5,  6,  7,  8,  9, 10, 11, 14, 13,  0, 14 },
                                    {  1,  1,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    {  4,  3,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 12, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 },
                                    { 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14 } };
 int car;
 int col;
 int estado = 0;

 int i = 0;

 do
 {
```

```
 car = fgetc(in);
 col = columna(car);
 estado = tabla[estado][col];

 if ( col != 11 )
 {
  buffer[i] = car;
  i++;
 }
}
while ( !estadoFinal(estado) && !(estado == 14) );

buffer[i] = '\0';

switch ( estado )
{
 case 2 : if ( col != 11 )
              {
               ungetc(car, in);
               buffer[i-1] = '\0';
              }
             return ID;
 case 4 : if ( col != 11 )
              {
               ungetc(car, in);
               buffer[i-1] = '\0';
              }
        return CONSTANTE;
 case 5 : return SUMA;
 case 6 : return RESTA;
 case 7 : return PARENIZQUIERDO;
 case 8 : return PARENDERECHO;
 case 9 : return COMA;
 case 10 : return PUNTOYCOMA;
 case 12 : return ASIGNACION;
 case 13 : return FDT;
 case 14 : return ERRORLEXICO;
 }
 return 0;
}
int estadoFinal(int e)
{
 if ( e == 0 || e == 1 || e == 3 || e == 11 || e == 14 ) return 0;
 return 1;
}
int columna(int c)
{
 if ( isalpha(c) ) return 0;
 if ( isdigit(c) ) return 1;
 if ( c == '+' ) return 2;
 if ( c == '-' ) return 3;
```

```
 if ( c == '(' ) return 4;
 if ( c == ')' ) return 5;
 if ( c == ',' ) return 6;
 if ( c == ';' ) return 7;
 if ( c == ':' ) return 8;
 if ( c == '=' ) return 9;
 if ( c == EOF ) return 10;
 if ( isspace(c) ) return 11;
 return 12;
}
/*************Fin Scanner*********************************************/
```

ASDR

inicio
leer(a);
leer(b);
c:=a+b;
escribir(c);
fin

El proceso de compilación se divide en dos partes principales: análisis y síntesis. El análisis, a su vez, consta de tres fases: Análisis Léxico, Análisis Sintáctico y Análisis Semántico. El documento describe la implementación de un Analizador Sintáctico Descendente Recursivo (ASDR), que es un tipo de analizador sintáctico que puede ser construido por un programador y produce una derivación por izquierda. Generando un (AAS)
En el ASDR, cada no terminal de la gramática del lenguaje tiene un Procedimiento de Análisis Sintáctico (PAS) asociado. Para procesar un terminal, se invoca a la función Match() con el terminal como argumento, validando que el token actual sea el esperado.

A continuación, se detalla el análisis del código de ejemplo inicio leer(a); leer(b);c=a+b; escribir(c);fin paso a paso, de acuerdo con la implementación en C.

Estructura de la ejecución
El programa principal (main) invoca a la función Objetivo(), que representa el axioma de la gramática.

El proceso de análisis es el siguiente:

Objetivo() llama a Programa().
Programa() llama a la función Comenzar().
Luego espera el token INICIO con Match(INICIO).
En este punto, el analizador léxico (scanner) lee la palabra "inicio" del código fuente y la clasifica como el token INICIO.
Match() valida que sea el token correcto.
Después de validar INICIO, Programa() llama a ListaSentencias().

Procesando las sentencias

La función ListaSentencias() procesa las sentencias dentro de un ciclo while. En cada iteración, llama a la función Sentencia().

Primera sentencia: leer(a);
Sentencia() lee el siguiente token, que es LEER.
Entra en el case LEER del switch.
Valida los tokens LEER y PARENIZQUIERDO con Match().
Llama a ListaIdentificadores().
ListaIdentificadores() invoca a Identificador().
Identificador() valida el token ID (a) con Match(ID) y llama a la rutina semántica ProcesarId() para registrar la variable 'a' en la tabla de símbolos.
De vuelta en ListaIdentificadores(), se llama a la rutina semántica Leer(reg) que genera la instrucción "Read a Entera".
Finalmente, se valida el token PUNTOYCOMA con Match(PUNTOYCOMA).

Segunda sentencia: leer(b);
El proceso se repite de manera idéntica al anterior. ListaSentencias() vuelve a llamar a Sentencia(), que procesa la entrada leer(b); y genera la instrucción "Read b Entera".

Tercera sentencia: c=a+b;
Sentencia() lee el token ID (c).
Entra en el case ID.
Llama a Identificador(&izq) para procesar la variable c.
La rutina semántica ProcesarId() la agrega a la tabla de símbolos y genera la instrucción "Declara c Entera".
Se valida el token ASIGNACION con Match(ASIGNACION).
Se llama a Expresion(&der).
Expresion() llama a Primaria(&operandoIzq).
Primaria() procesa el ID (a).
Expresion() lee el token SUMA. Llama a OperadorAditivo() para procesar +.
Expresion() llama a Primaria(&operandoDer) que procesa el ID (b).
Se invoca la rutina semántica GenInfijo() para generar la instrucción "Sumar a,b,Temp&1".
De vuelta en Sentencia(), se llama a la rutina semántica Asignar(izq, der), que genera la instrucción "Almacena Temp&1,c,".
Se valida el token PUNTOYCOMA con Match(PUNTOYCOMA).

Cuarta sentencia: escribir(c);
Sentencia() lee el token ESCRIBIR.
Entra en el case ESCRIBIR.
Valida los tokens ESCRIBIR y PARENIZQUIERDO con Match().
Llama a ListaExpresiones().
ListaExpresiones() invoca a Expresion() que, a su vez, llama a Primaria() para procesar el ID (c).
De vuelta en ListaExpresiones(), se llama a la rutina semántica Escribir(reg) que genera la instrucción "Write c,Entera,".
Se valida el token PUNTOYCOMA con Match(PUNTOYCOMA).

Finalización del programa

ListaSentencias() continúa su bucle, pero al no encontrar más sentencias válidas, retorna.
El control regresa a Programa(), que valida el token FIN con Match(FIN).
El control vuelve a Objetivo(), que valida el token FDT (Fin de Texto) con Match(FDT) y llama a la rutina semántica Terminar().
Terminar() genera la instrucción final "Detiene" para detener la ejecución.
De esta manera, el analizador sintáctico descendente recursivo procesa el programa paso a paso, utilizando una función para cada no terminal de la gramática, validando los tokens con la función Match() e invocando a las rutinas semánticas para generar el código intermedio a medida que avanza.

Análisis exhaustivo del compilador Micro: Uniendo la teoría y la implementación

Este compilador en un solo archivo es un ejemplo directo de cómo los conceptos teóricos del diseño de compiladores se materializan en código.
A través de un análisis detallado de cada componente de su código fuente, aquí se demuestra que la verdadera organización de un compilador no es física sino lógica, definida por la clara división del trabajo entre sus funciones y estructuras de datos.
Se examinan las interconexiones entre las fases, la abstracción del flujo de tokens, el uso de autómatas de estado finito y la integración de las acciones semánticas, proporcionando una guía definitiva que desmitifica la arquitectura del compilador Micro y, por extensión, los principios fundamentales de la construcción de compiladores.

1. **El marco conceptual del diseño de compiladores**

1.1 La esencia de un compilador: análisis y síntesis

Un compilador es, en su definición más fundamental, un programa de software diseñado para traducir el código fuente escrito en un lenguaje de programación a código objeto en un lenguaje de destino. Este proceso se divide conceptualmente en dos partes principales: el análisis y la síntesis. La fase de análisis se encarga de descomponer el programa fuente para comprender su estructura y significado, mientras que la fase de síntesis construye el programa destino basándose en esa comprensión.
El programa que se va a compilar se presenta al compilador como una secuencia de caracteres que termina con un centinela, en este caso, un punto (.). Esta dicotomía entre análisis y síntesis no solo define el propósito de un compilador, sino que también establece la columna vertebral lógica para su diseño.
La presentación del compilador Micro como un único archivo de código fuente en C es una decisión pedagógica deliberada. Mientras que los compiladores modernos a gran escala están altamente modularizados en múltiples archivos y bibliotecas para facilitar su desarrollo y mantenimiento, un enfoque monolítico obliga al estudiante a confrontar la separación de las preocupaciones lógicas (léxicas, sintácticas, semánticas) dentro de una única unidad física. Esta estructura simplifica la tarea de seguir el flujo de control de principio a fin, ya que se puede rastrear la ejecución a través de las llamadas a funciones sin la necesidad de navegar por un sistema de archivos complejo. La verdadera comprensión radica, entonces, en la capacidad de discernir los roles distintos de cada función y estructura de datos, un desafío que este informe se propone resolver.

1.2 Las fases analíticas de la compilación

La fase de análisis del compilador Micro está compuesta por tres subfases interdependientes: el análisis léxico, el análisis sintáctico y el análisis semántico.
El análisis léxico es la primera etapa. Su tarea es examinar la secuencia de caracteres de entrada, identificar las unidades léxicas (llamadas lexemas) y agruparlas en tokens o categorías léxicas significativas. Por ejemplo, en el lenguaje Micro, el lexema inicio se clasifica como el token INICIO, y 123 se clasifica como el token CONSTANTE.
Posteriormente, el análisis sintáctico toma el flujo de tokens producido por el analizador léxico y lo procesa para determinar si su secuencia cumple con las reglas gramaticales del lenguaje fuente.
Si bien el analizador sintáctico del Micro puede validar si las construcciones individuales son correctas, se señala que por sí solo no puede determinar si el programa completo es sintácticamente correcto, ya que depende de la fase semántica para los cheques de contexto.
Finalmente, el análisis semántico complementa la validación sintáctica. Las rutinas semánticas en el Micro cumplen dos funciones principales: verifican la semántica estática del programa (por ejemplo, que las variables se usen correctamente, aunque en este caso el único tipo de dato es entero) y, si la construcción es semánticamente correcta, inician la traducción a código intermedio.

En el compilador Micro, estas fases no están rígidamente separadas en módulos aislados. Por ejemplo, la función ProximoToken() del analizador sintáctico invoca al analizador léxico (scanner()), pero también realiza una búsqueda en la tabla de símbolos (Buscar()) para distinguir si un ID es una palabra reservada.
De manera similar, las funciones sintácticas (Sentencia, Expresion) invocan directamente a rutinas semánticas (Asignar, GenInfijo) para llevar a cabo la generación de código.
Esto revela que el compilador Micro utiliza una arquitectura de una sola pasada, donde el análisis léxico, el análisis sintáctico, el análisis semántico y la generación de código se ejecutan en un solo barrido del código fuente. Esta integración es la razón por la que las funciones del código están anidadas en un patrón de llamadas profundo y recursivo, con el análisis impulsando la síntesis de manera simultánea.

1.3 El lenguaje Micro: una descripción formal

El compilador Micro está diseñado para procesar un lenguaje formalmente definido. Las características clave de este lenguaje son:
Tipos de datos: El único tipo de dato permitido es el entero.
Identificadores y constantes: Los identificadores se declaran implícitamente, con una longitud máxima de 32 caracteres. Deben comenzar con una letra y pueden contener letras y dígitos. Las constantes son secuencias de dígitos.
Estructura del programa: El cuerpo de un programa está delimitado por las palabras reservadas inicio y fin.
Sentencias: Se admiten dos tipos de sentencias :
Asignación: ID := Expresión;
Entrada/Salida: leer (lista de IDs); o escribir (lista de Expresiones);
Cada sentencia debe terminar con un punto y coma (;).
Expresiones: Las expresiones son infijas y se construyen con identificadores, constantes y los operadores + y -. Los paréntesis están permitidos para alterar el orden de evaluación.
La gramática léxica y sintáctica del lenguaje está explícitamente definida en el material. Esta formalización es la base sobre la que se construyen los analizadores del compilador.

Por ejemplo, el analizador sintáctico descendente recursivo (ASDR) del Micro se implementa directamente a partir de las reglas de la gramática sintáctica.

2: El analizador léxico: De caracteres a tokens

2.1 El rol y la implementación del scanner()

El corazón del analizador léxico es la función scanner(), cuyo propósito es leer los caracteres del archivo de entrada (in) uno a uno y agruparlos en unidades léxicas (lexemas) para producir una representación de token. La función lee caracteres, los almacena en el buffer global y, al reconocer un lexema completo, devuelve un valor del tipo enumerado TOKEN que lo representa.

El analizador sintáctico no invoca directamente al scanner(). En su lugar, utiliza la función ProximoToken(), que actúa como una capa de abstracción. Esta indirection es una técnica de diseño fundamental, ya que desacopla la lógica del analizador sintáctico de los detalles de bajo nivel del escaneo de caracteres.

La función ProximoToken() implementa una forma de "evaluación perezosa" o "mecanismo de extracción de tokens": solo llama a scanner() para obtener un nuevo token si no hay uno disponible (flagToken es 0). Una vez obtenido, el token se almacena en la variable global tokenActual y flagToken se establece en 1.

La próxima vez que el analizador sintáctico necesite un token, ProximoToken() lo devuelve inmediatamente sin llamar al scanner(). Esto mejora la claridad del código del analizador sintáctico al permitirle concentrarse en las reglas gramaticales sin preocuparse por la gestión de la entrada de caracteres.

2.2 El autómata de estado finito y la tabla de transición

La función scanner() es la implementación de un autómata de estado finito determinista (AFD), un método clásico para la construcción de analizadores léxicos.

La lógica del AFD se codifica en una tabla de transición bidimensional llamada tabla. Esta matriz tiene 15 filas (para cada estado del autómata) y 13 columnas (para cada clase de carácter de entrada).

La función columna(c) es el puente entre el mundo de los caracteres sin procesar y la tabla de estados.

Asigna un índice de columna a cada carácter de entrada, categorizándolo como una letra, un dígito, un operador, un carácter de puntuación o un espacio en blanco.

El ciclo do-while de la función scanner() utiliza la tabla de transición para cambiar de estado en cada carácter. La función estadoFinal(e) comprueba si el estado actual es un estado terminal, lo que indica que se ha reconocido un lexema completo.

Una característica crucial en la implementación del scanner() es el uso de ungetc(car, in) cuando se reconoce un token. Este es un mecanismo para manejar un problema común en los AFD conocido como "sobrepaso". Por ejemplo, cuando el analizador reconoce el identificador a, lee el carácter a y luego el siguiente carácter (por ejemplo, un espacio o un operador) para determinar que el identificador ha terminado. Sin embargo, ese segundo carácter pertenece al siguiente lexema. El uso de ungetc devuelve el carácter al flujo de entrada, asegurando que el próximo escaneo comience en la posición correcta. Este detalle es vital para el funcionamiento correcto del analizador léxico carácter a carácter.

2.3 Estructuras de datos clave

El Micro define varias estructuras de datos para su analizador léxico y las fases posteriores. La enumeración TOKEN proporciona nombres simbólicos y legibles para cada tipo de token,

como ID, CONSTANTE, SUMA, y RESTA. Esto mejora la legibilidad y el mantenimiento del código al evitar el uso de números mágicos.
La tabla de símbolos (TS) es un arreglo de estructuras RegTS que se utiliza para almacenar los identificadores del programa. La tabla se inicializa con las palabras reservadas del lenguaje (inicio, fin, leer, escribir). Esta elección de diseño simplifica la lógica del escáner, ya que no necesita distinguir entre identificadores y palabras reservadas.
El scanner() simplemente clasifica todo lo que cumple con la regla de identificadores como ID, y la función ProximoToken() es la responsable de consultar la tabla de símbolos para ver si el ID escaneado es en realidad una palabra reservada.
Esta delegación de la lógica ilustra la fluidez con la que las fases del compilador se integran en el código.
A continuación, se presenta una tabla que consolida la información léxica, estableciendo un mapeo directo entre los lexemas del lenguaje, los nombres de los tokens y sus valores enumerados en la implementación:

inicio	INICIO	0	Delimita el inicio del programa.
fin	FIN	1	Delimita el final del programa.
leer	LEER	2	Palabra reservada entrada.
escribir	ESCRIBIR	3	Palabra reservada para salida.
	ID	4	Identificador de variable.
	CONSTANTE	5	Número entero.
(	PARENIZQUIERDO	6	Paréntesis de apertura.
)	PARENDERECHO	7	Paréntesis de cierre.
;	PUNTOYCOMA	8	Carácter de fin de sentencia.
,	COMA	9	Carácter separación de listas
:=	ASIGNACION	10	Operador de asignación.
+	SUMA	11	Operador aditivo.
-	RESTA	12	Operador aditivo.
Fin archivo	FDT	13	Indicador de fin de texto.
otro carácter	ERRORLEXICO	14	Error léxico.

3: El analizador sintáctico: Análisis descendente recursivo
3.1 La gramática de Micro y el análisis de arriba hacia abajo
La gramática sintáctica de Micro es una gramática independiente del contexto (GIC) que define la estructura jerárquica de los programas.
El analizador sintáctico del compilador Micro se basa en un método de análisis descendente recursivo (ASDR).
Este enfoque de "arriba hacia abajo" construye el árbol de análisis sintáctico a partir del axioma (símbolo de inicio) de la GIC. Una característica distintiva de un ASDR es su correspondencia directa entre cada no-terminal de la gramática (como <programa>, <sentencia>) y una función de análisis en el código.
A diferencia del análisis sintáctico ascendente (bottom-up), que utiliza una derivación a la derecha en orden inverso para "reducir" la cadena de entrada al axioma, el ASDR sigue una derivación a la izquierda.
Mostramos ests diferencia con un ejemplo de la cadena aabcdd, demostrando cómo la derivación por la izquierda sigue un proceso intuitivo de expansión desde el axioma hasta la cadena terminal.

Esta naturalidad es la razón por la que los ASDR pueden ser construidos manualmente por un programador, sin necesidad de herramientas especializadas como yacc.
En el compilador Micro, no se construye una estructura de datos explícita para el árbol de análisis sintáctico. En su lugar, el árbol está implícitamente representado por la secuencia de llamadas a funciones anidadas. La función Objetivo() llama a Programa(), que a su vez llama a ListaSentencias(), la cual llama a Sentencia(), y así sucesivamente.
La pila de llamadas de estas funciones de análisis refleja directamente la estructura del árbol, con cada llamada a una función correspondiendo a un nodo no-terminal en el árbol.
Esta elección de diseño simplifica enormemente la implementación y la hace más eficiente en cuanto al uso de memoria, ya que se evita la sobrecarga de construir una estructura de datos compleja.

3.2 La implementación en C: un análisis función por función
El análisis del código sintáctico revela una implementación directa de la gramática.

Objetivo(): Corresponde al axioma <objetivo> -> <programa> FDT.
Es el punto de entrada principal del analizador sintáctico y es la primera función que se invoca después de la inicialización.
Llama a Programa() y luego espera el token FDT (Fin de Texto), que es el centinela del programa fuente.

Programa(): Implementa la regla <programa> -> inicio <listaSentencias> fin. Se asegura de que el programa comience con INICIO, procese una lista de sentencias, y termine con FIN.

ListaSentencias(): Se encarga de la repetición {<sentencia>} de la gramática. Implementa esta repetición con un bucle while que invoca a la función Sentencia() repetidamente. El ciclo utiliza la función ProximoToken() para "mirar" hacia adelante y determinar si la próxima construcción es una sentencia válida.

Sentencia(): Utiliza una sentencia switch para manejar las tres posibles producciones de <sentencia>: asignación, lectura o escritura.
Según el tipo de token que ProximoToken() devuelva (ID, LEER o ESCRIBIR), se bifurca la ejecución a las funciones correspondientes.

Expresion() y Primaria(): Estas funciones demuestran la recursión. Expresion() maneja las sumas y restas, y llama a Primaria() para procesar los operandos.
Primaria() es recursiva ya que puede manejar una expresión entre paréntesis, que a su vez llama a Expresion().

El funcionamiento de todo el analizador se basa en un contrato de dos partes: ProximoToken() y Match().
Las funciones de análisis sintáctico utilizan ProximoToken() para inspeccionar el siguiente token en el flujo de entrada y tomar una decisión, como en el switch de Sentencia().
Una vez que se ha tomado una decisión, se llama a Match(t) para consumir ese token del flujo.
Match() no solo consume el token, sino que también valida que el token actual sea el que el analizador sintáctico esperaba, lanzando un ErrorSintactico() si no es así.

Este patrón de "inspeccionar y consumir" es el motor que garantiza que el procesamiento del flujo de tokens se mantenga sincronizado con las reglas de la gramática.

3.3 Funciones de manejo de errores

El manejo de errores en este compilador pedagógico es intencionalmente rudimentario.
Las funciones ErrorLexico() y ErrorSintactico() simplemente imprimen un mensaje a la consola y no implementan mecanismos de recuperación de errores.
Si un token inesperado aparece en cualquier punto del proceso de análisis, el compilador se detiene.
Este diseño simplificado es apropiado para un ejemplo de aprendizaje, ya que el enfoque está en la corrección del análisis más que en la robustez frente a errores.

4: El analizador semántico y la generación de código intermedio

4.1 La tabla de símbolos y las rutinas semánticas

La tabla de símbolos (TS) es una estructura de datos crucial para el análisis semántico. Almacena información sobre los identificadores del programa. En Micro, cada entrada en la TS contiene el nombre del identificador y un xcódigo que indica si es una palabra reservada o un identificador de variable.
El compilador gestiona la tabla de símbolos a través de tres funciones principales:
Buscar(id, TS, &t): Busca un identificador en la tabla de símbolos para determinar si ya existe.
Colocar(id, TS): Agrega un nuevo identificador a la tabla si no está presente.
Chequear(s): Combina las dos funciones anteriores. Si una cadena no se encuentra en la tabla, la agrega y, si es el nombre de una variable, genera una instrucción de declaración.
Las rutinas semánticas del compilador Micro se invocan directamente desde el analizador sintáctico.
Por ejemplo, ProcesarId() es una rutina semántica llamada desde la función sintáctica Identificador(), y ProcesarId() a su vez llama a Chequear().
Esta arquitectura demuestra una de las características clave del compilador Micro: un diseño de una sola pasada. No hay una fase de análisis de símbolos separada y dedicada. En cambio, el compilador realiza la verificación y la gestión de la tabla de símbolos "al vuelo" a medida que el programa es analizado.
Actividad Este enfoque fusiona la fase de análisis con la de síntesis, lo que reduce la complejidad de la implementación y el número de pasadas sobre el código fuente.

4.2 La generación de código intermedio (Generar())

El compilador Micro no produce código máquina directamente. En su lugar, la fase de síntesis genera una representación de código intermedio para una máquina virtual hipotética.
Este código es una serie de instrucciones simples, generalmente en formato de tres direcciones. La función Generar(co, a, b, c) es el motor de esta fase. Toma un código de operación (co) y hasta tres operandos (a, b, c) y los formatea como una línea de texto de salida.
Cada rutina semántica invoca a Generar() con una instrucción específica:

1. Leer(in) genera una instrucción Read.
2. Escribir(out) genera una instrucción Write.
3. El Asignar(izq, der) genera una instrucción Almacena.
4. GenInfijo(e1, op, e2) genera instrucciones Sumar o Restar.

5. Chequear(s) genera la instrucción Declara cuando se encuentra un nuevo identificador.
6. Terminar() genera la instrucción Detiene, que finaliza la ejecución del programa compilado.

4.3 Traducción de expresiones (GenInfijo())

La función GenInfijo() es un ejemplo particularmente interesante de la generación de código. Su tarea es traducir expresiones infijas (como a + b) a código intermedio. Para manejar expresiones complejas, como x + y - z, GenInfijo() no produce una sola instrucción. En cambio, genera variables temporales para almacenar los resultados parciales de las subexpresiones.

La función utiliza una variable estática numTemp para generar nombres de variables temporales únicos, como Temp&1, Temp&2, etc.. Para una expresión como x + y - z, se realizarían las siguientes acciones:

La subexpresión x + y se evalúa. GenInfijo() genera una instrucción Sumar x, y, Temp&1.

La subexpresión Temp&1 - z se evalúa. GenInfijo() genera una instrucción Restar Temp&1, z, Temp&2.

El uso de variables temporales es una técnica de generación de código muy sofisticada que descompone expresiones complejas en una secuencia de instrucciones de tres direcciones simples y fáciles de ejecutar por la máquina virtual.

Esta técnica demuestra que, a pesar de su naturaleza pedagógica, el compilador Micro incorpora principios de ingeniería de compiladores muy sólidos.

5**: Una visión holística: Integración y flujo de datos**

5.1 El flujo de ejecución del compilador

El punto de entrada físico del compilador Micro es la función main(). Su papel es el de un orquestador.

Se encarga del manejo de argumentos de la línea de comandos, la apertura del archivo de código fuente y, lo más importante, inicia el proceso de compilación al invocar a la función Objetivo().

A partir de esta llamada inicial, el control se transfiere a las funciones de análisis sintáctico, que a su vez se comunican con el analizador léxico y las rutinas semánticas en un flujo de control entrelazado.

El compilador Micro depende en gran medida de un conjunto de variables globales para la comunicación entre sus componentes (in, buffer, tokenActual, flagToken, TS).

Aunque el uso extensivo de variables globales se considera a menudo una mala práctica en el desarrollo de software a gran escala, en este caso, es una elección de diseño intencionada para la enseñanza.

Estas variables actúan como un sistema nervioso central, proporcionando un canal de comunicación directo y sencillo entre las fases lógicamente separadas. Por ejemplo, la función scanner() escribe en el buffer global, el cual es luego leído por las rutinas semánticas como ProcesarId() y ProcesarCte().

De manera similar, ProximoToken() y Match() manipulan las variables globales tokenActual y flagToken para controlar el avance entre el analizador léxico y el sintáctico.

Este diseño simplifica el rastreo del flujo de datos para un estudiante, ya que no necesita seguir complejas estructuras de datos pasadas por referencia o punteros

5.2 Un ejemplo práctico: el rastreo de la compilación

Para ilustrar cómo los componentes del compilador Micro trabajan en conjunto, se presenta a continuación un rastreo paso a paso de la compilación de un programa de ejemplo.

Programa de ejemplo en Micro:

```
inicio
 leer(x,y);
 z := x+y;
 escribir(z);
fin.
```

Flujo de la compilación:

Inicio: main() abre el archivo fuente y llama a Objetivo().

Análisis Sintáctico: Objetivo() invoca a Programa(), que luego llama a Comenzar(), una rutina semántica de inicialización.

Análisis Léxico/Sintáctico: Programa() llama a Match(INICIO). ProximoToken() invoca a scanner() para obtener el token INICIO, el cual es consumido

Lista de Sentencias: Programa() invoca a ListaSentencias(), la cual entra en un ciclo. En cada iteración, ProximoToken() obtiene el siguiente token (LEER, ID, ESCRIBIR) y llama a Sentencia().

Procesando leer(x,y);:
Sentencia() identifica el token LEER y llama a Match(LEER).
Match(PARENIZQUIERDO) consume el (.
ListaIdentificadores() se invoca. ProximoToken() obtiene ID (x).
Identificador(&reg) llama a Match(ID).
ProcesarId() es llamada. Análisis Semántico: Chequear("x") verifica que x no está en la tabla de símbolos, lo añade y llama a Generar("Declara", "x", "Entera", "").
Generación de Código: Se imprime la instrucción Declara x,Entera,.

Análisis Semántico: Leer(reg) llama a Generar("Read", "x", "Entera", "").
Generación de Código: Se imprime la instrucción Read x,Entera,.
ProximoToken() obtiene ,. El bucle de ListaIdentificadores() continúa. El mismo proceso ocurre para y.
Match(PARENDERECHO) consume). Match(PUNTOYCOMA) consume ;.
Procesando z := x+y;:
Sentencia() identifica el token ID (z).
Identificador(&izq) llama a ProcesarId(). Análisis Semántico: Chequear("z") se ejecuta, declara z y genera la instrucción Declara z,Entera,.
Match(ASIGNACION) consume :=.
Expresion(&der) se invoca para procesar x+y.
LaPrimaria(&operandoIzq) procesa x. Análisis Semántico: ProcesarId() llama a Chequear("x"), que ya existe, no se genera una nueva declaración.
ProximoToken() obtiene SUMA.
OperadorAditivo() procesa +.

Primaria(&operandoDer) procesa y. Análisis Semántico: ProcesarId() llama a Chequear("y"), que ya existe.

Análisis Semántico/Generación de Código: GenInfijo() se invoca. Genera una variable temporal, Temp&1, y genera la instrucción Sumar x,y,Temp&1. Chequear("Temp&1") la declara, generando Declara Temp&1,Entera,.
Expresion() retorna el registro semántico de Temp&1.

Análisis Semántico/Generación de Código: Asignar(izq, der) se invoca. Llama a Generar("Almacena", "Temp&1", "z", ""), generando Almacena Temp&1,z,.
Match(PUNTOYCOMA) consume ;.

Procesando escribir(z);:
Sentencia() identifica ESCRIBIR.
Match(ESCRIBIR) consume el token.
Match(PARENIZQUIERDO) consume (.
ListaExpresiones() se invoca.
Expresion(&reg) procesa z.

Análisis Semántico/Generación de Código: Escribir(reg) llama a Generar("Write", "z", "Entera", ""), generando Write z,Entera,.
Match(PARENDERECHO) consume). Match(PUNTOYCOMA) consume ;.

Finalización: ListaSentencias() finaliza su bucle, Match(FIN) consume el token FIN. Objetivo() llama a Match(FDT), que consume el centinela, y luego a Terminar(), que genera la instrucción Detiene.
Salida de código intermedio final:

```
Declara x,Entera,
Read x,Entera,
Declara y,Entera,
Read y,Entera,
Declara z,Entera,
Declara Temp&1,Entera,
Sumar x,y,Temp&1
Almacena Temp&1,z,
Write z,Entera,
Detiene ,,
```

Este rastreo muestra cómo la compilación es una progresión continua a través del código fuente, con cada componente activando el siguiente y contribuyendo al resultado final, todo dentro de una sola unidad de código.

Puntos clave y recomendaciones

6.1 Resumen de los hallazgos

El análisis del compilador Micro responde de manera exhaustiva a la pregunta inicial. El código fuente de un compilador no solo puede agruparse en un único documento, sino que hacerlo en un contexto de aprendizaje revela la esencia de su organización: una estructura intrínsecamente lógica, no física.

El compilador Micro logra esto a través de una división clara de funciones para el análisis léxico, sintáctico y semántico, incluso cuando estas funciones residen en el mismo archivo y se entrelazan en una arquitectura de una sola pasada.
Las decisiones de diseño, como la implementación de un analizador léxico basado en un AFD con una tabla de transición, la utilización de un analizador sintáctico descendente recursivo que representa el árbol de análisis de forma implícita y la integración de rutinas semánticas y de generación de código "al vuelo", son testimonio de la solidez conceptual de su arquitectura. La dependencia de variables globales y la generación de variables temporales para expresiones son ejemplos de cómo se han aplicado técnicas de diseño de compiladores para crear un sistema simple pero funcional y educativo.

6.2 El compilador Micro como herramienta pedagógica

El compilador Micro es un recurso de aprendizaje invaluable. Su simplicidad lo hace accesible, mientras que su funcionalidad completa permite a los estudiantes ver cómo los conceptos abstractos de las gramáticas y autómatas se traducen en un programa ejecutable que realiza una tarea compleja. La correspondencia directa entre las reglas de la gramática y las funciones de análisis sintáctico proporciona un marco de estudio claro.
La falta de complejidad en el manejo de errores o la gestión de múltiples pasadas permite que el estudiante se concentre en el núcleo del proceso de traducción.

6.3 Recomendaciones para trabajos futuros

Para avanzar más allá de la base establecida por el compilador Micro, se podrían explorar varias extensiones que reflejen un sistema más avanzado:
Gestión de tipos de datos: Implementar un sistema de tipos más rico que incluya, por ejemplo, números de punto flotante o cadenas de texto, requiriendo un análisis semántico más sofisticado para la verificación de tipos.
Estructuras de control: Añadir sentencias de control de flujo como if y while al lenguaje. Esto exigiría una modificación de la gramática sintáctica y la adición de rutinas semánticas para la generación de código con saltos condicionales.
Mejora del manejo de errores: En lugar de detener el compilador ante el primer error, implementar mecanismos de recuperación de errores que permitan al compilador continuar el análisis y reportar múltiples problemas.
Optimización del código intermedio: Desarrollar una fase de optimización que mejore el código intermedio generado antes de la síntesis final, por ejemplo, eliminando variables temporales innecesarias.
Generación de código máquina: Modificar la función Generar() para producir código en un lenguaje de ensamblaje real para una arquitectura de procesador específica, en lugar de un código intermedio para una máquina virtual abstracta.
Estas posibles mejoras demuestran que, aunque el compilador Micro es un sistema autocontenido y funcional, también sirve como un trampolín para la exploración de conceptos de compiladores más avanzados, ofreciendo un camino claro para un estudio más profundo de la materia.

Lex y yacc

Mientras que el analizador descendente recursivo del compilador Micro fue construido manualmente por un programador, los analizadores ascendentes (conocidos como bottom-up) suelen requerir la ayuda de herramientas especializadas como yacc.

1. El rol de lex y yacc

lex: Es un generador de analizadores léxicos . Se le proporcionaría la gramática léxica del lenguaje Micro, que define los patrones para cada tipo de token (ID, CONSTANTE, INICIO, SUMA, etc.) utilizando expresiones regulares . lex procesaría estas reglas y generaría un programa en C, típicamente llamado yylex(), que lee el código fuente del programa Micro y lo convierte en una secuencia de tokens .

yacc: Es un generador de analizadores sintácticos ascendentes que utiliza el método LALR (Lookahead Left-to-right Rightmost derivation) . Se le proporcionaría la gramática sintáctica de Micro . A cada una de las reglas de la gramática, se le adjuntarían "acciones" en C que se ejecutarían cada vez que el analizador ascendente reconozca una construcción gramatical . El programa principal del compilador llamaría a la función yyparse() generada por yacc, la cual, a su vez, invocaría a yylex() para obtener los tokens .

2. El proceso de análisis ascendente para el ejemplo

El análisis ascendente (también llamado shift-reduce) es la inversa de una derivación a la derecha. En lugar de partir del axioma y expandir las reglas, parte de la secuencia de tokens y las "reduce" de vuelta al axioma, construyendo el árbol de análisis desde las hojas hacia la raíz.

Para el ejemplo inicio leer(a); leer(b);c=a+b; escribir(c);fin, el proceso sería el siguiente:

Análisis Léxico (lex): lex leería el código fuente y produciría un flujo de tokens para yacc: INICIO, LEER, PARENIZQUIERDO, ID (a), PARENDERECHO, PUNTOYCOMA, ID (b), ASIGNACION, etc.

Análisis Sintáctico (yacc): El analizador yacc comenzaría a leer este flujo de tokens, aplicando una secuencia de operaciones shift y reduce para construir el árbol de análisis implícitamente:

Shift: El analizador consume un token de la entrada y lo empuja a una pila.

Reduce: Cuando el analizador encuentra en la cima de la pila una secuencia de tokens y/o no terminales que coincide con el lado derecho de una regla de la gramática, reemplaza esa secuencia por el no terminal del lado izquierdo de la regla. En este punto, se ejecutarían las acciones en C asociadas a la regla.

Ejecución de las acciones semánticas:

Cuando el analizador reconociera la secuencia ID := Expresion;, ejecutaría la acción asociada a la regla <sentencia> de asignación. Esta acción llamaría a la rutina semántica Asignar() para generar la instrucción de código intermedio "Almacena"

De manera similar, para la expresión a+b, el analizador de yacc reconocería la secuencia de tokens ID, SUMA, ID.

Cuando esta secuencia se corresponde con la regla <expresion> -> <primaria> <operadorAditivo> <primaria>, se ejecutaría la acción adjunta para llamar a la rutina semántica GenInfijo().

Esta rutina generaría una variable temporal y la instrucción de código intermedio Sumar.

Las rutinas semánticas Leer() y Escribir() se invocarían al reconocer las reglas de gramática para leer y escribir, respectivamente,
El proceso continuaría de esta manera, "reduciendo" las sentencias individuales a <sentencia>, luego las sentencias agrupadas a <listaSentencias>, y finalmente, al reconocer INICIO <listaSentencias> FIN, se reduciría al axioma <programa>, confirmando que el programa completo es sintácticamente correcto. De esta forma, el análisis ascendente impulsado por lex y yacc podría generar el mismo código intermedio que el analizador descendente recursivo.

Implementar en C
La implementación del analizador en C se basa en la idea de que cada no terminal en la gramática del lenguaje tiene su propia función que lo procesa. El analizador léxico se encarga de leer el código fuente y devolver los tokens.
A continuación, se describen los componentes clave del programa en C:
Variables Globales: Se utilizan variables globales para mantener el estado del analizador.
* token: Almacena el token actual que está siendo procesado.
* in: Puntero al archivo de entrada (.m o .micro) que contiene el código fuente.
* buffer: Un array de caracteres que contiene el lexema del token actual.

Análisis Léxico (scanner): Esta función se encarga de leer el código fuente y devolver el siguiente token. Generalmente se implementa como una máquina de estados finita que reconoce los diferentes tipos de tokens como ID, CONSTANTE, INICIO, etc. En el código proporcionado, la función scanner lee caracteres, los clasifica según su tipo (letra, dígito, etc.) y usa una matriz de transición (tabla) para determinar el estado siguiente. Al llegar a un estado final, se devuelve el tipo de token correspondiente.

Función Match(expected_token): Esta función auxiliar es fundamental para el proceso. Su única responsabilidad es verificar si el token actual coincide con el expected_token. Si es así, llama a la función scanner() para obtener el siguiente token. Si no coinciden, genera un error sintáctico.

Rutinas Semánticas: Son funciones C que se encargan de generar el código intermedio (cuádruplas) a medida que el analizador reconoce las estructuras sintácticas correctas. Por ejemplo, la rutina Generar() crea una cuádrupla con un operador y sus operandos, mientras que ProcesarId() y ProcesarCte() gestionan la tabla de símbolos.

Funciones de los No Terminales: Son las funciones que implementan la gramática del lenguaje.
* Objetivo(): La función principal que inicia el análisis.
* Programa(): Llama a las funciones para procesar el token INICIO, la ListaSentencias y el token FIN.
* ListaSentencias(): Un bucle while que llama repetidamente a la función Sentencia() mientras haya sentencias por procesar.
* Sentencia(): Utiliza una sentencia switch para decidir qué tipo de sentencia se está analizando (LEER, ESCRIBIR, o asignación con un ID). Cada case llama a las funciones correspondientes para los no terminales que le siguen.

* Expresion(), Primaria(), OperadorAditivo(): Estas funciones procesan las expresiones aritméticas y sus componentes.

El programa principal (main) abre el archivo de entrada, inicializa la tabla de símbolos, llama al scanner para obtener el primer token, y luego invoca a Objetivo() para comenzar el análisis.

Proceso de ejecución
Para ejecutar el analizador, se deben seguir los siguientes pasos:
Compilar el código fuente: Asumiendo que todos los archivos C (.c) y de cabecera (.h) están en el mismo directorio, se utiliza un compilador de C como GCC. El comando de compilación sería similar a este:

```
gcc -o compilador main.c scanner.c rutinas_semanticas.c ...
```

Aquí, compilador es el nombre del archivo ejecutable, y se listan todos los archivos fuente necesarios.

Crear un archivo de código fuente del lenguaje Micro: Se crea un archivo de texto con extensión .m o .micro (por ejemplo, programa.micro) que contiene el código que se quiere analizar. Por ejemplo, el código del documento:

```
 inicio
leer(a);
leer(b);
c = a + b;
escribir(c);
fin
```

Ejecutar el programa compilado: El ejecutable del compilador toma como argumento el nombre del archivo de código fuente del lenguaje Micro. El comando de ejecución es:

```
./compilador programa.micro
```

Al ejecutar este comando, el analizador:
* Leerá el archivo programa.micro.
* El scanner irá obteniendo los tokens uno a uno.
* La función Objetivo() y las siguientes (como Programa(), ListaSentencias(), etc.) irán procesando el flujo de tokens.
* Si la sintaxis del programa es correcta, las rutinas semánticas se activarán para generar el código intermedio (generalmente escrito en un archivo de salida).
* Si hay algún error sintáctico, la función Match() lo detectará y el programa mostrará un mensaje de error y terminará.

De esta manera, el compilador procesará el archivo de entrada y producirá un archivo de salida con el código intermedio si la sintaxis del programa de entrada es correcta.

Anatomía de un Compilador: Del Diseño Manual a la Generación Automática del Lenguaje Micro con Lex y Yacc

1 Introducción: Fundamentos de la Compilación y lateoria de Lenguajes

1.1. Contexto

La creación de un programa fuente para ser procesado por las herramientas lex y yacc refleja un momento crucial en la comprensión de la teoría de los compiladores.
El material de referencia disponible aborda la construcción de un compilador de manera manual, con un enfoque particular en el análisis sintáctico descendente recursivo (ASDR) del Lenguaje Micro.
Este enfoque contrasta fundamentalmente con la metodología de generación automática propuesta por herramientas como lex y yacc, las cuales, de manera subyacente, se basan en un análisis sintáctico ascendente.
El propósito de este apunte es servir como un puente conceptual, traduciendo la comprensión de la implementación manual del Lenguaje Micro a un paradigma de ingeniería de software más abstracto y automatizado.
Se examina la sinergia entre estas herramientas y se proporciona un estudio de caso detallado para ilustrar cómo la gramática del Lenguaje Micro puede ser implementada de manera eficiente y robusta a través de la generación de código.

1.2. Fases de la Compilación y la Interdependencia de los Analizadores

El proceso de compilación se divide en dos partes principales: el análisis y la síntesis.
La fase de análisis se encarga de descomponer el programa fuente en sus partes constituyentes y de verificar su validez estructural y semántica.
Esta fase está compuesta por tres subfases interdependientes: el análisis léxico, el análisis sintáctico y el análisis semántico.
El análisis léxico es la primera fase, donde el scanner lee una secuencia de caracteres y los agrupa en lexemas, los cuales son clasificados en categorías léxicas o tokens.
Este proceso no detecta la corrección de las construcciones en su totalidad, sino que prepara una secuencia de unidades lógicas para la siguiente fase. El análisis sintáctico, o parsing, toma esta secuencia de tokens y determina si la estructura del programa es sintácticamente correcta de acuerdo con las reglas de una gramática.
Durante este proceso, se construye implícita o explícitamente un árbol de análisis sintáctico.
El análisis semántico complementa al sintáctico, verificando la corrección de las construcciones que el parser no puede validar, como la consistencia de tipos de datos o la correcta declaración de variables. Las rutinas semánticas también tienen la función de realizar la traducción y generar el código intermedio para la fase de síntesis.

1.3. La Gramática del Lenguaje Micro

Para la implementación con lex y yacc, la gramática del Lenguaje Micro sirve como la especificación formal del lenguaje. Esta gramática se divide en dos componentes principales: la gramática léxica y la gramática sintáctica.
La Gramática Léxica define las unidades atómicas del lenguaje, los tokens :
* <token> -> uno de <identificador> <constante> <palabraReservada> <operadorAditivo> <asignación> <carácterPuntuación>

* <identificador> -> <letra> {<letra o dígito>}
* <constante> -> <dígito> {<dígito>}
* <letra o dígito> -> uno de <letra> <dígito>
* <letra> -> una de a-z A-Z
* <dígito> -> uno de 0-9
* <palabraReservada> -> una de inicio fin leer escribir
* <operadorAditivo> -> uno de + -
* <asignación> -> :=
* <carácterPuntuación> -> uno de ( ) , ;

La Gramática Sintáctica define la estructura jerárquica de las sentencias y expresiones :

* <objetivo> -> <programa> fdt
* <programa> -> inicio <listaSentencias> fin
* <listaSentencias> -> <sentencia> {<sentencia>}
* <sentencia> -> <identificador> := <expresión> ; | leer (<listaIdentificadores>) ; | escribir (<listaExpresiones>) ;
* <listaIdentificadores> -> <identificador> { , <identificador>}
* <listaExpresiones> -> <expresión> { , <expresión>}
* <expresión> -> <primaria> {<operadorAditivo> <primaria>}
* <primaria> -> <identificador> | <constante> | (<expresión>)

2. Análisis Léxico Automatizado con Lex

2.1. La Lógica del Análisis Léxico y el Automóvil Finito

El análisis léxico tiene la responsabilidad de leer el flujo de caracteres del programa fuente y producir una secuencia de tokens.

El material del Lenguaje Micro muestra una implementación de este proceso a través de una máquina de estados finitos codificada manualmente en C. La función scanner() utiliza una tabla de transición (tabla) y funciones auxiliares como columna() y estadoFinal() para simular el comportamiento de un autómata finito determinista (AFD).

El proceso implica leer un carácter (fgetc(in)), determinar la columna correspondiente en la tabla (columna(car)), y actualizar el estado actual (estado = tabla[estado][col]) hasta alcanzar un estado final.

Este enfoque manual, si bien didáctico, es susceptible a errores y altamente laborioso.

La principal ventaja de una herramienta como lex (o su versión moderna flex) radica en que automatiza completamente la creación de este AFD.

En lugar de codificar la tabla de transiciones y la lógica de estados manualmente, el programador simplemente especifica los patrones de los tokens utilizando expresiones regulares. lex se encarga de generar el código C (lex.yy.c) que implementa el autómata y la lógica de transición de forma optimizada, liberando al desarrollador de una tarea repetitiva y compleja.

La tabla de transición del scanner del Lenguaje Micro es, de hecho, la representación explícita en C del autómata que lex generaría automáticamente a partir de las reglas de gramática léxica.

2.2. Creación del Archivo de Especificación Lex (micro.l)

Un archivo de especificación de lex (.l) está estructurado en tres secciones, separadas por delimitadores %%.

La primera sección, Declaraciones, contiene el código C inicial, como directivas #include, macros o variables globales, que deben estar encerradas entre %{ y %}.
En este caso, la inclusión del archivo de cabecera generado por yacc (y.tab.h) es fundamental, ya que este archivo define los nombres de los tokens como constantes numéricas.
La segunda sección, Reglas, es donde se definen los patrones de expresiones regulares y las acciones asociadas a cada uno. Las acciones son fragmentos de código C que se ejecutan cuando el scanner reconoce un patrón.
Para el Lenguaje Micro, las reglas de la gramática léxica se traducen a esta sección.
Por ejemplo, la regla para identificador se traduce a una expresión regular que coincide con una letra seguida de cero o más letras o dígitos.
La tercera y última sección, Subrutinas, puede contener cualquier código C adicional, como la función main() o la función yywrap(). Aunque lex puede incluir versiones por defecto de estas funciones, es una buena práctica definirlas explícitamente.

A continuación, se presenta una aproximación del contenido del archivo micro.l, que ilustra cómo los conceptos de la gramática léxica se mapean a las expresiones regulares de lex.

```
/* micro.l - Analizador léxico para el Lenguaje Micro */
%{
#include "y.tab.h"
#include <stdio.h>
#include <string.h>
%}

/* Definiciones de expresiones regulares */
letra [a-zA-Z]
digito [0-9]
identificador {letra}({letra}|{digito})*
constante {digito}+

%%
/* Reglas de token y acciones asociadas */

"inicio"       return INICIO;
"fin"          return FIN;
"leer"         return LEER;
"escribir"     return ESCRIBIR;

{identificador} {
    /* Aquí se podría llamar a una rutina semántica
     * para buscar en la tabla de símbolos y retornar
     * ID o una palabra reservada, como se hace en el
     * código C manual del documento. */
    yylval.str = strdup(yytext);
    return ID;
}
{constante}    {
```

```
    yylval.str = strdup(yytext);
    return CONSTANTE;
}
":="          return ASIGNACION;
"+"           return SUMA;
"-"           return RESTA;
"("           return PARENIZQUIERDO;
")"           return PARENDERECHO;
","           return COMA;
";"           return PUNTOYCOMA;

[ \t\n]       /* Ignorar espacios en blanco */ ;

.             return ERRORLEXICO;

%%
/* Funciones auxiliares */

int yywrap() {
    return 1;
}
```

2.2.1: Nombres de Tokens y su Mapeo

El material proporciona un enum que define los tokens numéricos del Lenguaje Micro. yacc, al generar el archivo y.tab.h, asigna automáticamente valores enteros a los tokens declarados, garantizando la coherencia entre el scanner y el parser. Esta es una simplificación significativa en comparación con la definición manual de enum en el código en C.

inicio	INICIO	0
fin	FIN	1
leer	LEER	2
escribir	ESCRIBIR	3
id	ID	4
constante	CONSTANTE	5
(	PARENIZQUIERDO	6
)	PARENDERECHO	7
;	PUNTOYCOMA	8
,	COMA	9
:=	ASIGNACION	10
+	SUMA	11
-	RESTA	12
Fin de archivo	FDT	13
Carácter inválido	ERRORLEXICO	14

3. Análisis Sintáctico: Del Árbol de Derivación a la Reducción con Yacc

3.1. El Paradigma de Análisis Sintáctico Ascendente

El análisis sintáctico valida la estructura de la secuencia de tokens. Existen dos estrategias principales: el análisis descendente (top-down) y el análisis ascendente (bottom-up). El material ejemplifica el análisis descendente recursivo (ASDR), que construye un árbol de análisis desde el axioma de la gramática hacia las hojas (los tokens) a través de una derivación por izquierda.
Este método se implementa manualmente con procedimientos recursivos en C que corresponden a cada no-terminal de la gramática.
En contraste, yacc genera un analizador LALR (Lookahead Left-to-right Rightmost derivation), que utiliza una estrategia ascendente o de reducción-desplazamiento.
Este método trabaja en orden inverso a la derivación por derecha, "reduciendo" las secuencias de tokens hasta alcanzar el axioma de la gramática.
La diferencia fundamental entre un analizador descendente y uno ascendente radica en la gestión de la memoria de control.
Un analizador descendente se basa en la pila de llamadas del sistema (recursión) para simular el árbol de derivación. Un analizador ascendente, en cambio, utiliza una pila de estados explícita para seguir las producciones de la gramática y decidir cuándo "desplazar" un token a la pila o "reducir" una secuencia de símbolos a un no-terminal.
Esta capacidad de yacc para manejar una pila lo hace particularmente adecuado para analizar estructuras anidadas como paréntesis, algo que un simple analizador léxico sin pila no puede hacer.

3.2. Creación del Archivo de Gramática Yacc (micro.y)

El archivo de gramática yacc (.y) también se divide en tres secciones.
La primera sección, Declaraciones, contiene las definiciones para el analizador. Aquí se declaran los tokens no literales con la directiva %token, se especifica el símbolo de inicio de la gramática (<objetivo>) con %start , y se puede definir una unión (%union) para manejar los diferentes tipos de datos que los símbolos pueden tener, lo cual es vital para las acciones semánticas.
La segunda sección, Reglas de Gramática, contiene las producciones con acciones semánticas asociadas, que son fragmentos de código C encerrados entre llaves {}.
La implementación manual del compilador de Micro invoca a rutinas semánticas a través de llamadas a funciones como Asignar(), Leer() o Escribir().
En yacc, estas llamadas se incrustan directamente en las reglas de la gramática, y la herramienta permite acceder a los valores de los símbolos de la producción a través de la notación $n y asignar un valor al no-terminal del lado izquierdo con $$.
La tercera sección, Programas, incluye las funciones auxiliares y la función main() que inicia el proceso de análisis llamando a yyparse().
A continuación, se presenta una aproximación del contenido del archivo micro.y.

```
/* micro.y - Analizador sintáctico para el Lenguaje Micro */
%{
#include <stdio.h>
#include "lex.yy.c"
#include "rutinas.c" /* Archivo con rutinas semánticas */

extern FILE *in;
extern char buffer;
extern TOKEN tokenActual;
extern RegTS TS;
```

```
// Union para manejar atributos de expresiones
%union {
    char *str;
}

// Declaraciones de tokens (terminales)
%token INICIO FIN LEER ESCRIBIR ID CONSTANTE PARENIZQUIERDO
PARENDERECHO PUNTOYCOMA COMA ASIGNACION SUMA RESTA FDT

// Símbolo de inicio de la gramática
%start objetivo

// Tipos de atributos para no-terminales
%type <str> expresion primaria identificador

%%
/* Reglas de la gramática y acciones semánticas */

objetivo : programa FDT
{
    Terminar();
};

programa : INICIO listaSentencias FIN
{
    Comenzar();
};

listaSentencias : sentencia

| listaSentencias sentencia ;

sentencia : identificador ASIGNACION expresion PUNTOYCOMA
{
    Asignar($1, $3);
}

| LEER PARENIZQUIERDO listaIdentificadores PARENDERECHO PUNTOYCOMA
{
    // Las rutinas de 'leer' se ejecutan en listaIdentificadores
}

| ESCRIBIR PARENIZQUIERDO listaExpresiones PARENDERECHO PUNTOYCOMA
{
    // Las rutinas de 'escribir' se ejecutan en listaExpresiones
};
```

```
listaIdentificadores : identificador
{
   Leer($1);
}

| listaIdentificadores COMA identificador
{
   Leer($3);
};

listaExpresiones : expresion
{
   Escribir($1);
}

| listaExpresiones COMA expresion
{
   Escribir($3);
};

expresion : primaria

| expresion SUMA primaria
{
   $$ = GenInfijo($1, "+", $3);
}

| expresion RESTA primaria
{
   $$ = GenInfijo($1, "-", $3);
};

primaria : identificador

| CONSTANTE
| PARENIZQUIERDO expresion PARENDERECHO ;

identificador : ID
{
   $$ = $1;
};

%%
/* Código de usuario */

int yyerror(char const *s) {
   fprintf(stderr, "Error Sintáctico: %s\n", s);
   return 0;
```

```
}

int main(int argc, char **argv) {
   if (argc!= 2) {
      printf("Debe ingresar el nombre del archivo fuente.\n");
      return -1;
   }
   if ((in = fopen(argv, "r")) == NULL) {
      printf("No se pudo abrir el archivo fuente.\n");
      return -1;
   }
   return yyparse();
}
```

4. El Flujo de Trabajo Completo y la Sinergia Lex-Yacc

4.1. El Ecosistema de la Generación de Compiladores

lex y yacc no son herramientas que operan de forma aislada; su poder reside en su sinergia. Juntos, forman un sistema coherente donde lex maneja la fase léxica y yacc se encarga de la sintáctica, comunicándose a través de un protocolo bien definido.

La función principal del parser generado por yacc es yyparse(). Este procedimiento invoca automáticamente al analizador léxico yylex(), generado por lex, cada vez que necesita el siguiente token de la entrada. Esta colaboración de "solicitar-y-obtener" es el motor del proceso de compilación.

4.2. El Proceso de Generación y Compilación

El proceso de construcción del compilador a partir de los archivos de especificación (.l y .y) sigue un flujo de trabajo estándar en la línea de comandos

* **Generación del Parser:** Se invoca yacc con la bandera -d para procesar el archivo de gramática, por ejemplo, yacc -d micro.y. Este comando produce dos archivos C cruciales: y.tab.c, que contiene el código del analizador sintáctico (yyparse), y y.tab.h, un archivo de cabecera que define los tokens como constantes enteras.
* **Generación del Scanner**: Se procesa el archivo de especificación de lex con el comando lex micro.l. Este comando genera un archivo C llamado lex.yy.c, que contiene el código del analizador léxico (yylex).
* **Compilación y Enlazado**: Finalmente, se compilan y enlazan los archivos generados con un compilador de C, como gcc: gcc y.tab.c lex.yy.c -o micro_compilador. Es posible que se requiera enlazar con la biblioteca de lex con la bandera -ll. Este paso produce el ejecutable final, micro_compilador.

4.3. La Integración de la Tabla de Símbolos

La Tabla de Símbolos (TS) es una estructura de datos vital para la compilación, utilizada principalmente por las rutinas semánticas. En el enfoque manual del Lenguaje Micro, las funciones Buscar() y Colocar() son responsables de gestionar la TS, que predefine las palabras reservadas y almacena los identificadores a medida que se encuentran.

En el contexto de lex y yacc, la TS se integra como una estructura de datos global. La función yylex(), al reconocer un patrón de identificador, puede llamar a una rutina auxiliar que busca el lexema en la TS y determina si es una palabra reservada o un nuevo identificador. De manera similar, las acciones semánticas dentro de la gramática de yacc

pueden invocar a rutinas que interactúan con la TS, como Chequear(buffer) del código manual, para agregar un identificador si no existe y generar el código de declaración apropiado

5. Estudio de Caso y Análisis Comparativo Detallado

5.1. Comparación de Arquitecturas: Código Manual vs. Código Generado

El código de referencia del Lenguaje Micro representa una implementación de compilador de "diseño manual", donde cada fase y cada estructura se codifican explícitamente en C. El analizador léxico se basa en una tabla de transición, y el analizador sintáctico utiliza un conjunto de procedimientos recursivos para cada no-terminal.

En contraste, el enfoque con lex y yacc se basa en la "generación de código". El programador no escribe el scanner ni el parser, sino que especifica las reglas de la gramática en un lenguaje de alto nivel. Las herramientas traducen esta especificación en un código C optimizado que realiza el análisis.

5.2. Ventajas y Desventajas de cada Enfoque

El enfoque manual ofrece un control granular y una comprensión profunda de cada paso del proceso de compilación, lo que lo hace ideal para fines didácticos. Sin embargo, este método es inherentemente propenso a errores, consume mucho tiempo y es difícil de mantener y escalar para lenguajes más complejos.

La generación automática con lex y yacc ofrece una serie de ventajas significativas. Reduce drásticamente el tiempo de desarrollo y la probabilidad de errores en la implementación de los autómatas. La abstracción de las reglas de gramática en un formato legible mejora la mantenibilidad del proyecto. Además, los generadores de parsers como yacc están diseñados para manejar eficientemente gramáticas más complejas, resolviendo automáticamente conflictos de shift/reduce o reduce/reduce y permitiendo la definición de precedencia y asociatividad de operadores (%left, %right). La principal desventaja es una curva de aprendizaje inicial para dominar la sintaxis de las herramientas y un menor control sobre el código de bajo nivel generado, que puede parecer una "caja negra" difícil de depurar.

Comparativa de Estrategias de Análisis Sintáctico

Característica
- Análisis Sintáctico Descendente Recursivo (ASDR)
- Análisis Sintáctico Ascendente (LALR)

Metodología
- Diseño manual
- Generador de parser

Estrategia de Análisis
- Top-down (de la raíz a las hojas)
- Bottom-up (de las hojas a la raíz) |

Implementación
- Funciones recursivas en C
- Código C autogenerado (yyparse)

Tipo de Derivación
- Por izquierda
- Por derecha, en orden inverso

Mecanismo de Pila
- Pila de llamadas del sistema (implícita)

Pila de estados explícita
Robustez de la Gramática
Requiere gramáticas LL(1)
Acepta un subconjunto más amplio de gramáticas (LALR(1))
Flexibilidad de Diseño
Control completo sobre la lógica interna
Abstracción de bajo nivel, con reglas de alto nivel
Manejo de Operadores
Requiere refactorización de gramáticas para precedencia
Directivas de la herramienta (%left, %right) |

Una gramática LL(1) es un tipo de gramática libre de contexto que es ideal para ser analizada por un analizador sintáctico descendente predictivo. El nombre lo explica todo:
* L: El analizador procesa la entrada de izquierda a derecha.
* L: Genera la derivación más a la izquierda de la cadena.
* (1): Utiliza un lookahead de un solo símbolo (es decir, mira un token por delante) para decidir qué producción de la gramática debe aplicar.
En el contexto del "Compilador Micro" que hemos analizado, el Analizador Sintáctico Descendente Recursivo (ASDR) es un ejemplo de un analizador de este tipo. El hecho de que cada no-terminal (<programa>, <sentencia>) se corresponda con una función en el código del compilador es un rasgo característico de un analizador LL(1).
La diferencia con otras herramientas como Yacc es que estas, en general, implementan analizadores LR (Left-to-right, Rightmost derivation), que construyen el árbol de análisis "de abajo hacia arriba" (ascendente) en lugar de "de arriba hacia abajo" como lo hace un LL(1). Esto hace que las gramáticas LL(1) sean un subconjunto de las gramáticas LR, lo que significa que las gramáticas LR pueden analizar un conjunto más amplio de lenguajes de programación.

Una gramática LALR(1) ("Lookahead Left-to-right Rightmost derivation") es un tipo de gramática libre de contexto que se utiliza para construir analizadores sintácticos. Es una versión más eficiente del analizador LR(1), que es más potente que un analizador LL(1).
Características Principales
* LALR(1) es un método de análisis sintáctico ascendente, lo que significa que construye el árbol de análisis "de abajo hacia arriba", desde las hojas hasta la raíz.
* LR(1) es una familia de analizadores sintácticos que incluye SLR(1), LR(1) canónico y LALR(1). El analizador LALR(1) combina estados similares del analizador LR(1) canónico para crear tablas de análisis más pequeñas y eficientes sin sacrificar la capacidad de análisis.
* Aunque es menos potente que el LR(1) canónico, LALR(1) es suficientemente poderoso para la gran mayoría de los lenguajes de programación, por lo que es el método más utilizado en la práctica para la construcción de analizadores sintácticos, como los que se generan con herramientas como Yacc y Bison.
* A diferencia de las gramáticas LL(1), las gramáticas LALR(1) no tienen problemas con la recursividad izquierda.
El video a continuación te brinda una explicación más detallada sobre las diferencias entre las gramáticas LL(1), LR(1) y LALR(1).
LR(1) y LALR(1) | Análisis Sintáctico Ascendente

6. Conclusiones y Perspectivas Futuras

La transición de un compilador manual a uno generado con lex y yacc representa una evolución natural en la ingeniería de lenguajes. El análisis de la implementación manual del Lenguaje Micro demuestra que los principios fundamentales de la compilación no cambian, sino que se elevan a un nivel de abstracción superior. Las herramientas como lex y yacc no eliminan la necesidad de comprender la teoría, sino que la hacen más accesible y productiva al automatizar las tareas más tediosas y propensas a errores.

El Lenguaje Micro, con su gramática simple y su único tipo de dato, es un excelente caso de estudio para esta transición.

El paradigma de lex/yacc demuestra una escalabilidad inherente que es crucial para manejar lenguajes más complejos con múltiples tipos de datos, estructuras de control (if/else, while) y expresiones aritméticas con precedencia y asociatividad.

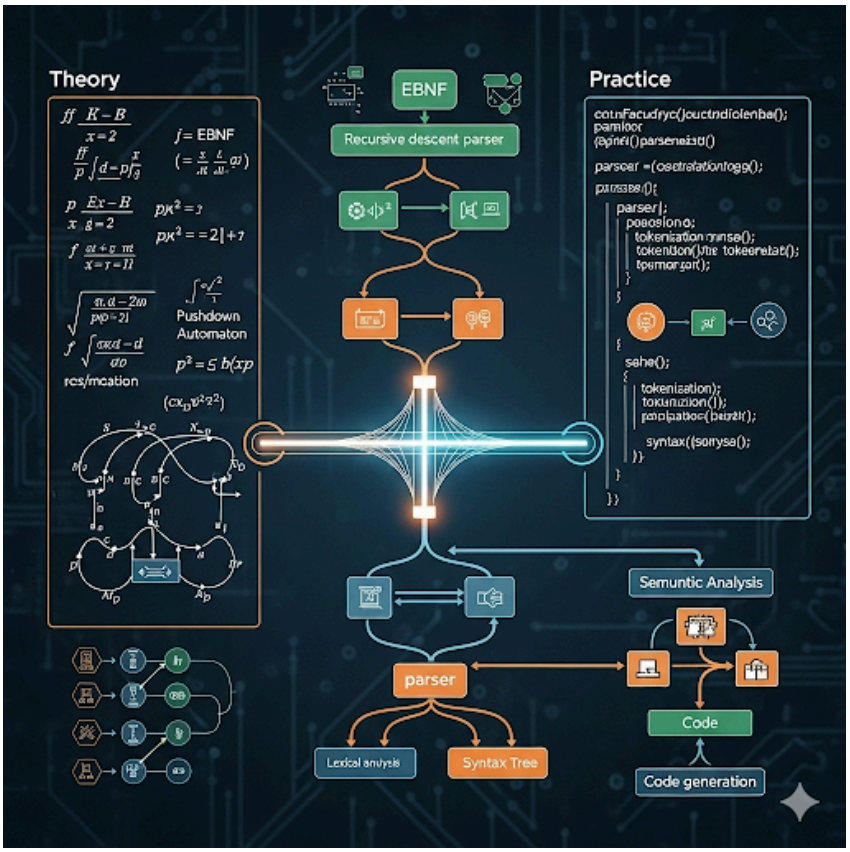
El uso de directivas de la herramienta para definir estas reglas simplifica significativamente el diseño del lenguaje.

En la actualidad, las implementaciones más utilizadas de estas herramientas son flex (un reemplazo más rápido de lex) y bison (la contraparte de yacc en el proyecto GNU).

MEl entendimiento de los principios subyacentes de lex y yacc sigue siendo fundamental para quienes desean dominar la ingeniería de lenguajes, sin importar si utilizan estas herramientas clásicas o alternativas más modernas como ANTLR o PLY. La capacidad de escribir una gramática y dejar que un generador de analizadores construya el parser es una habilidad invaluable en el desarrollo de software.

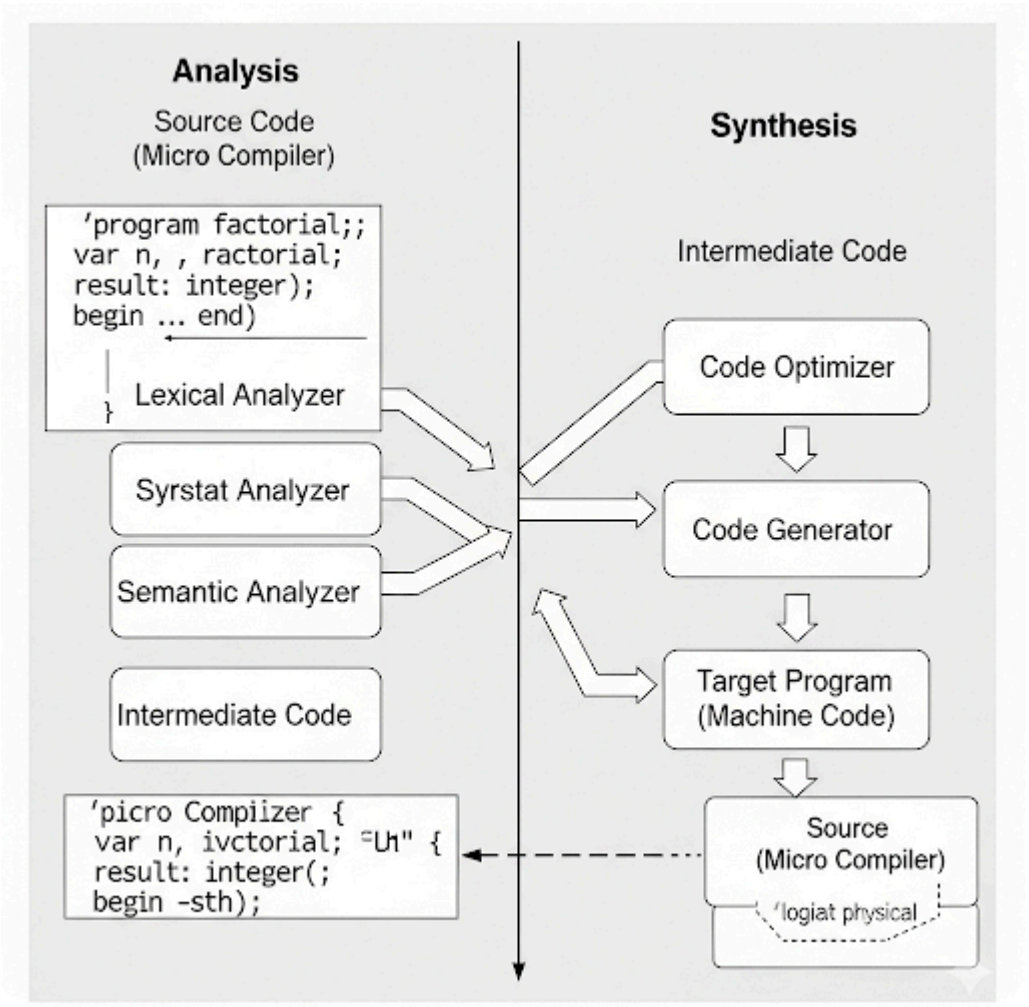
SINTESIS

Análisis Detallado del Compilador Micro: Uniendo Teoría y Práctica: Una visión de la arquitectura y la implementación de un analizador descendente recursivo.



¿Qué es un compilador?
Un programa que traduce código fuente a código objeto.

Las dos fases principales de la compilación:
Análisis: Descompone el programa fuente para comprender su estructura y significado.
Síntesis: Construye el programa de destino basándose en el análisis.
Objetivo de este análisis: Demostrar que la organización de un compilador es lógica, no física, a través del análisis del código fuente del compilador Micro.

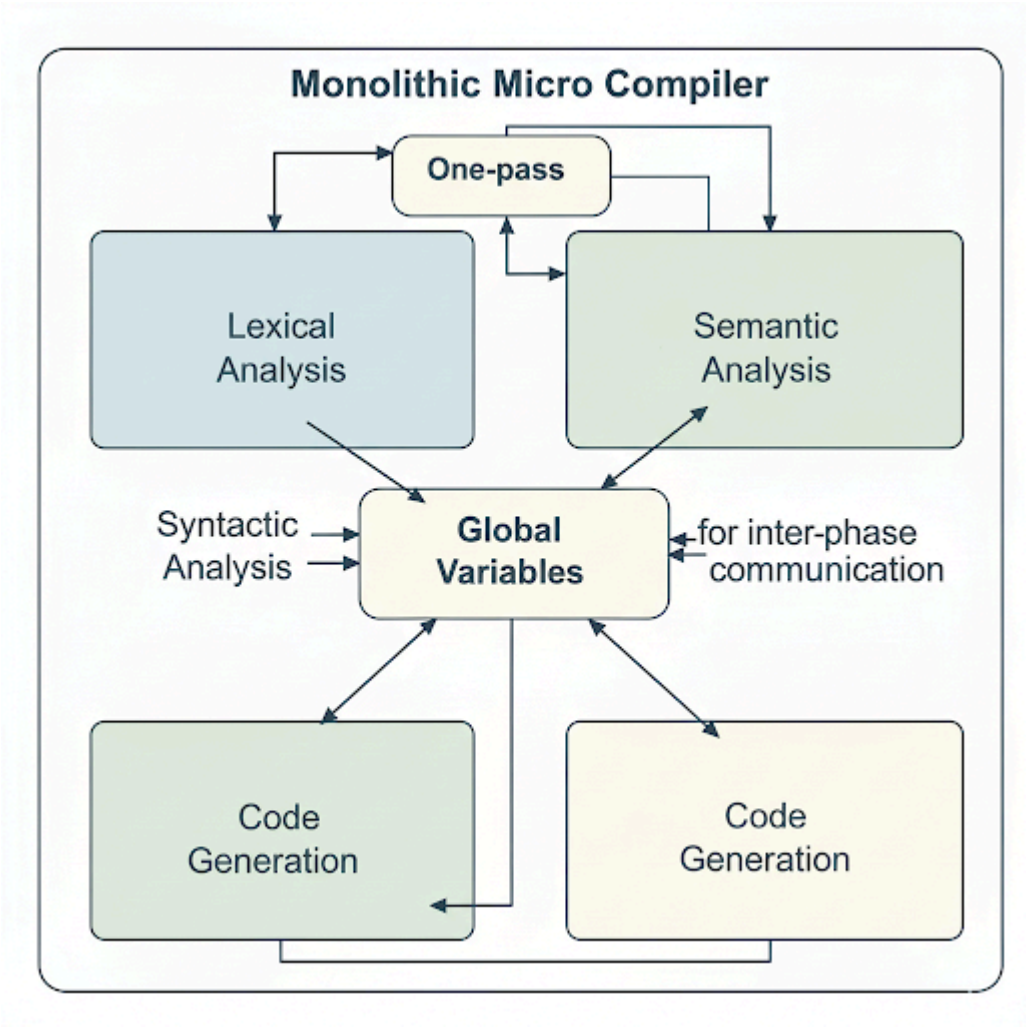


La Arquitectura del Compilador Micro

* Diseño de una sola pasada: Las fases de análisis léxico, sintáctico, semántico y la generación de código se ejecutan en un solo barrido del código fuente.

* Estructura Monolítica: Un único archivo de código en C que obliga a distinguir entre las preocupaciones lógicas de cada fase.

* Comunicación entre fases: Se utilizan variables globales para un flujo de datos sencillo y directo entre los componentes.



El Lenguaje Micro

* Características Clave:

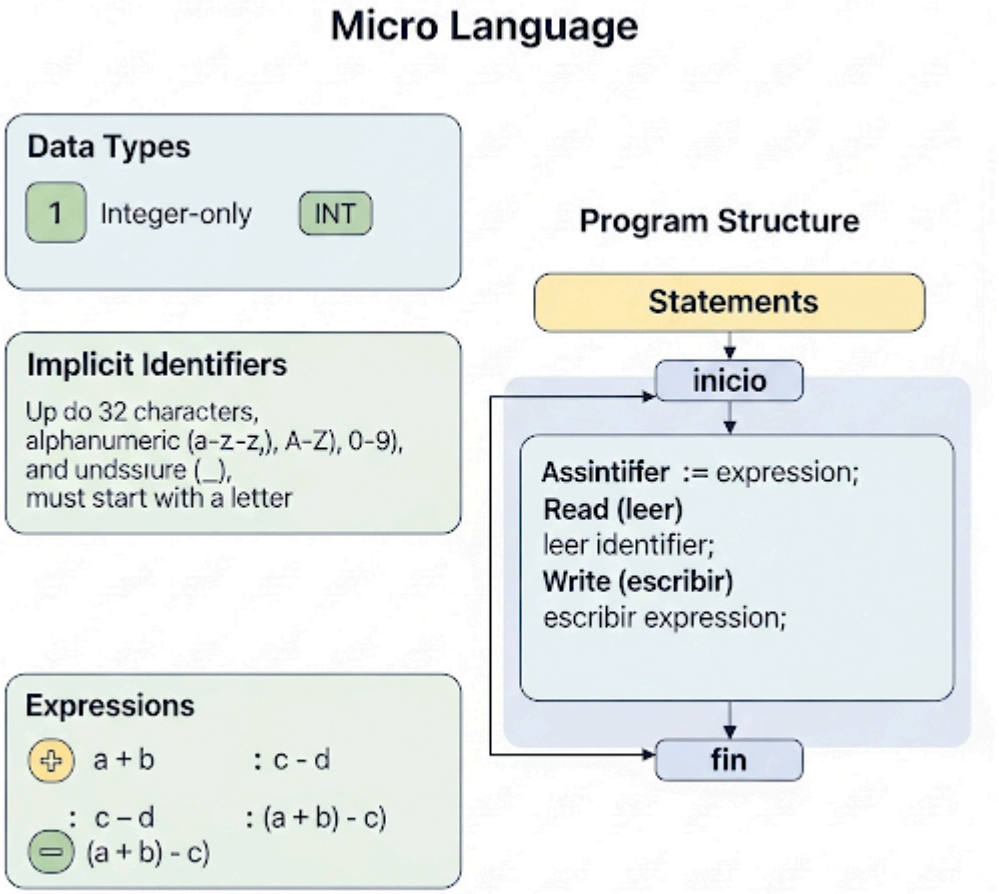
*Tipos de datos: Solo se permiten números enteros.

* Identificadores: Se declaran implícitamente, con un máximo de 32 caracteres, comenzando con una letra.

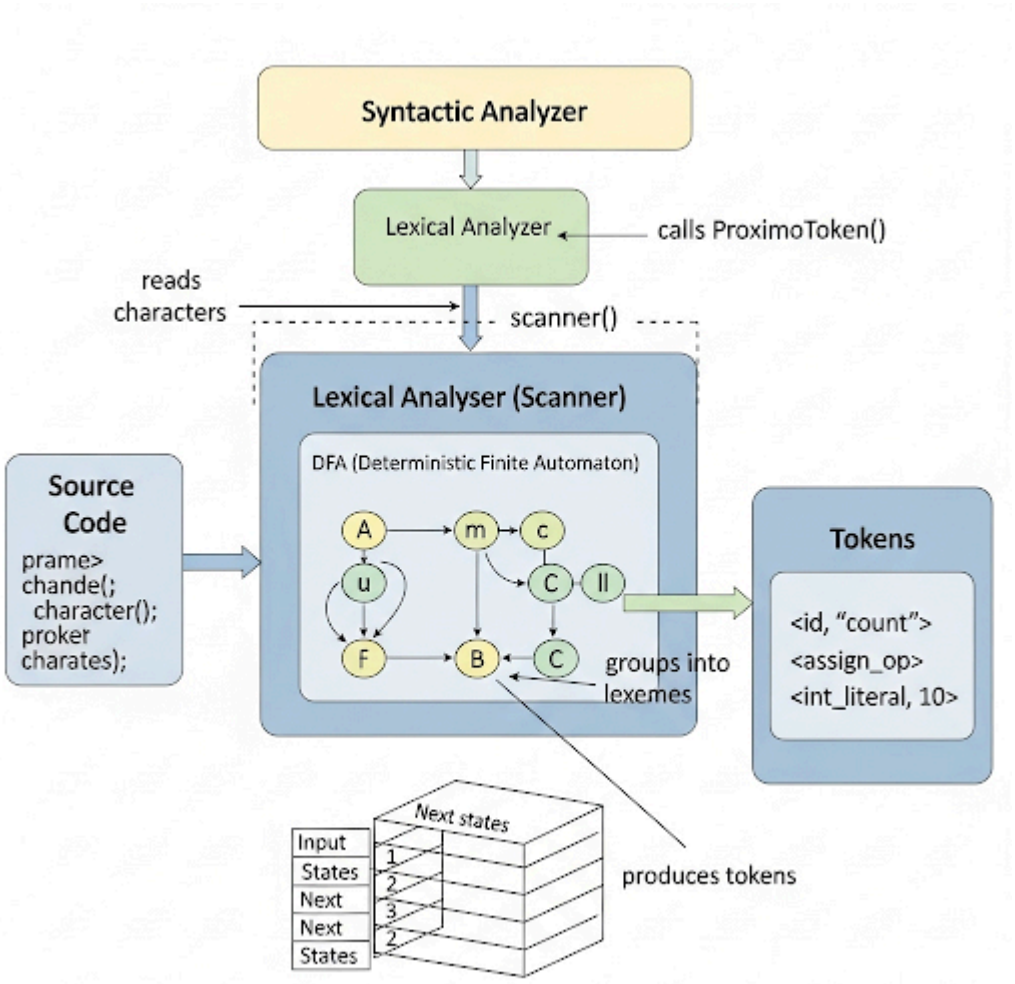
* Estructura del programa: Delimitado por las palabras reservadas inicio y fin.

* Sentencias: Asignación (:=), entrada (leer), y salida (escribir).
Cada una termina con un punto y coma (;).

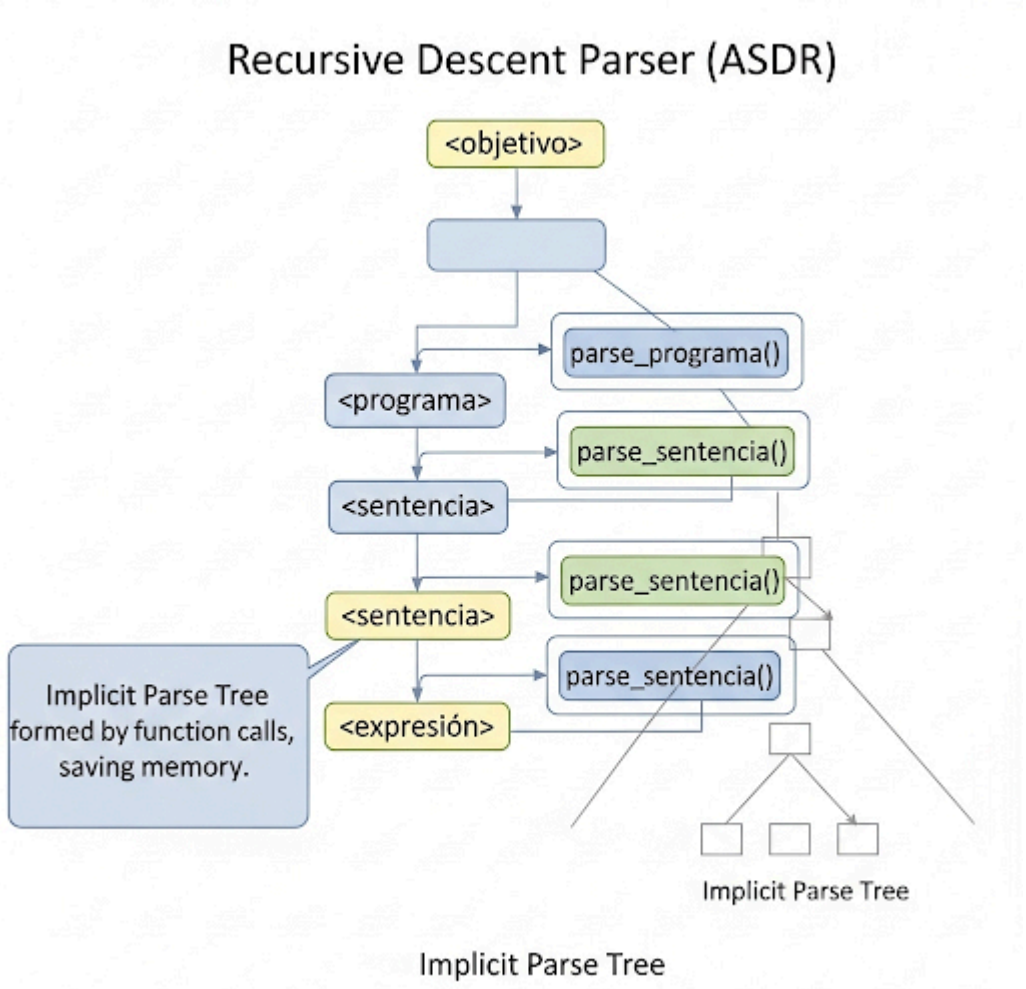
* [cite_start]Expresiones: Operaciones infijas con + y -, y uso de paréntesis.



Análisis Léxico (Scanner)
 * Función principal: scanner(). [cite_start]Lee caracteres, los agrupa en unidades léxicas (lexemas) y produce tokens.
 * Mecanismo: Implementación de un autómata de estado finito determinista (AFD). [cite_start]La lógica está codificada en una tabla de transición bidimensional.
 * Abstracción: El analizador sintáctico no llama a scanner() directamente, sino a ProximoToken(), lo que desacopla las dos fases.



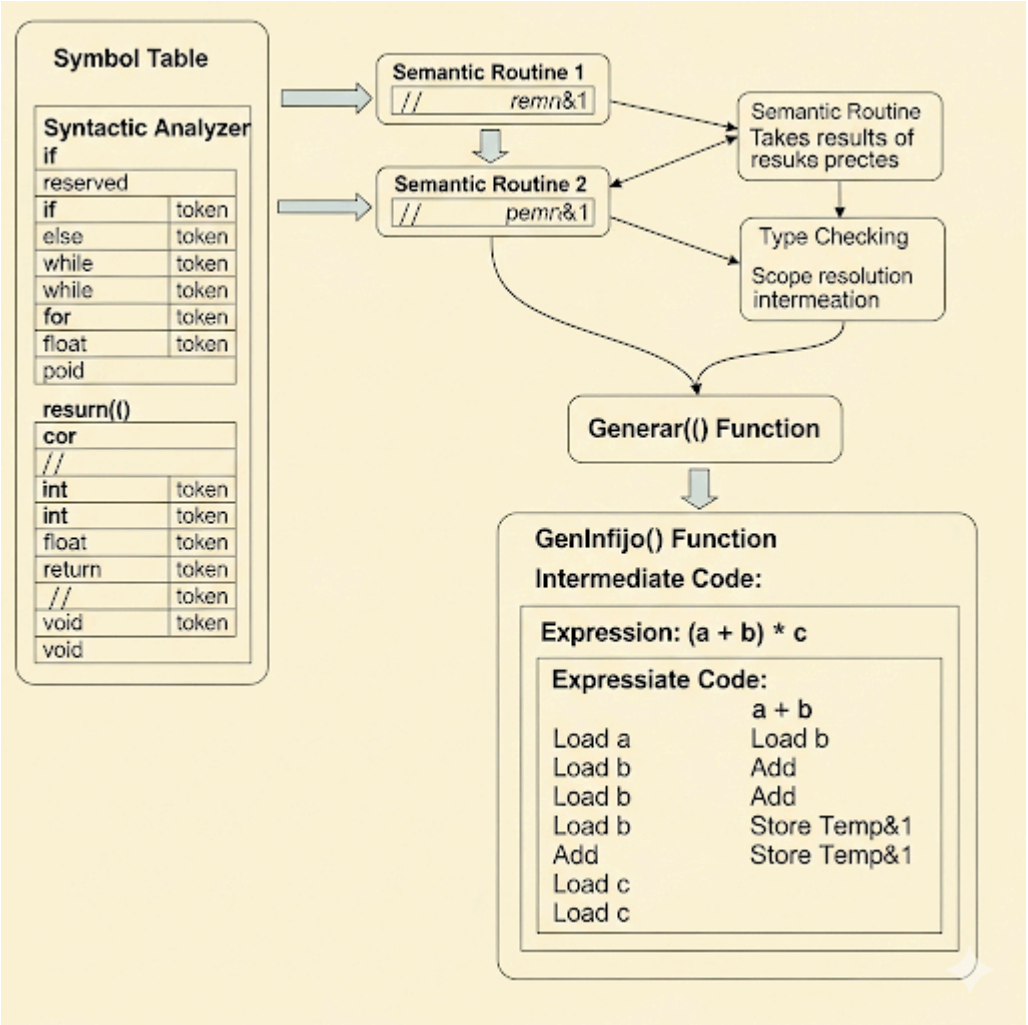
Análisis Sintáctico (ASDR)
 * Método: Análisis Descendente Recursivo (ASDR).
 * Principio "de arriba hacia abajo": Construye el árbol de análisis desde el axioma de la gramática (<objetivo>).
 * Implementación: Hay una correspondencia directa entre cada no-terminal de la gramática (ej., <programa>, <sentencia>) y una función en el código.
 *Árbol implícito: La estructura del árbol de análisis está representada por la pila de llamadas de las funciones anidadas, lo que ahorra memoria.



Implicit Parse Tree

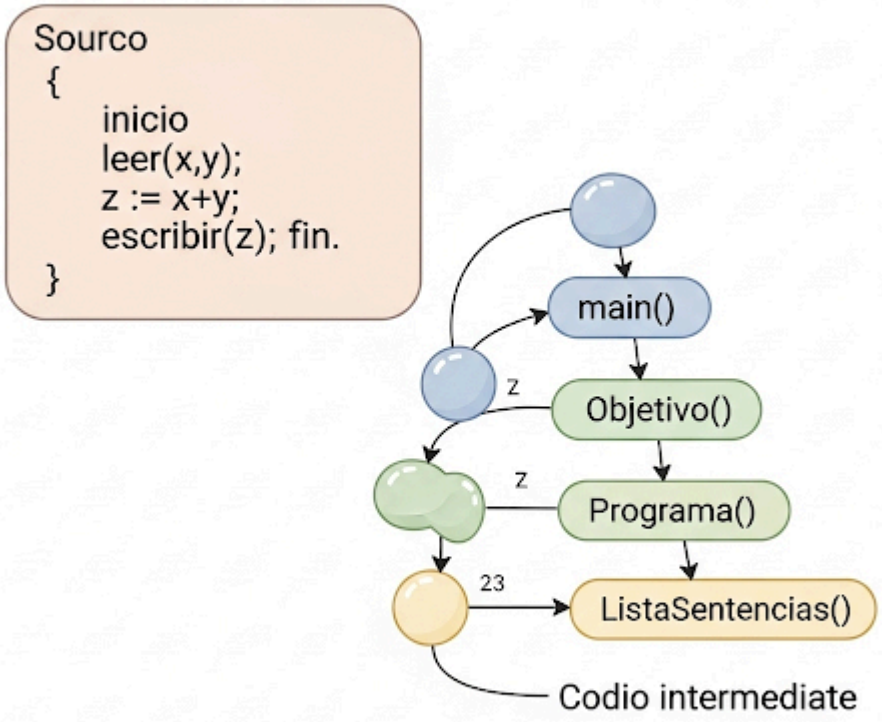
Análisis Semántico y Generación de Código Intermedio

* Tabla de Símbolos (TS): Almacena información sobre los identificadores. Se inicializa con palabras reservadas.
* Rutinas Semánticas: Se invocan directamente desde el analizador sintáctico para realizar la verificación semántica y generar el código.
* Generación de Código: La función Generar() produce una representación de código intermedio para una máquina virtual hipotética.
* Manejo de expresiones: GenInfijo() traduce expresiones complejas (ej., a+b) a instrucciones simples utilizando variables temporales (ej., Temp&1).



Rastreo de un Ejemplo

* Código de ejemplo: inicio leer(x,y); z := x+y; escribir(z); fin..
* Flujo paso a paso:
  *main() llama a Objetivo().
  *Objetivo() llama a Programa(), que procesa inicio y fin.
  *ListaSentencias() procesa cada sentencia (leer, :=, escribir).
  *El analizador léxico y el sintáctico trabajan juntos para consumir tokens.
  *Las rutinas semánticas (ej., Chequear(), Asignar(), Escribir()) generan las instrucciones de código intermedio (Declara, Read, Almacena, Write, Detiene) a medida que el análisis avanza.



Conclusiones y Futuras Mejoras

*Conclusión: El compilador Micro demuestra que los principios de la compilación son lógicos y pueden implementarse de forma simple pero robusta.

* Posibles mejoras:

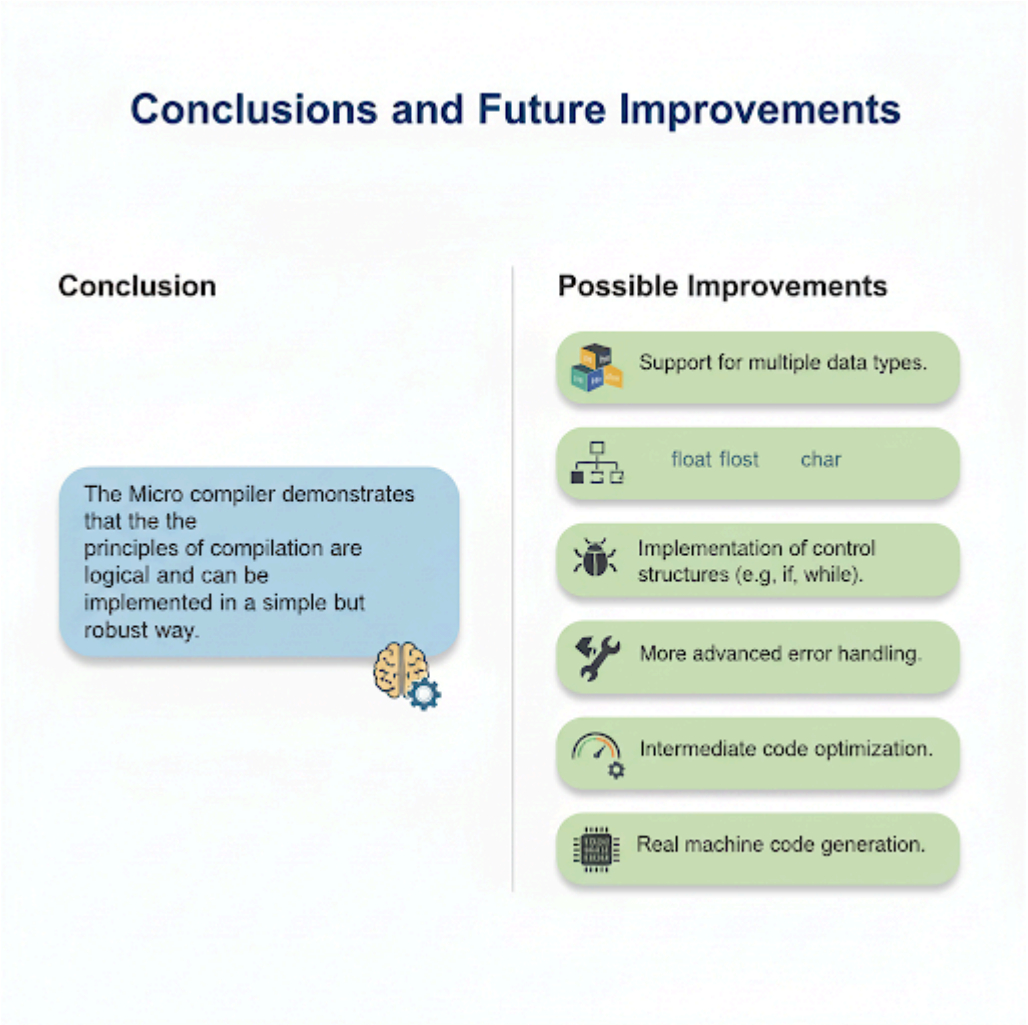
*Soporte para múltiples tipos de datos.

*Implementación de estructuras de control (ej., if, while).

*Manejo de errores más avanzado para no detener la compilación ante el primer error.

*Optimización del código intermedio.

*Generación de código máquina real.



Herramientas Relacionadas: Lex y Yacc
*Lex: Generador de analizadores léxicos que utiliza expresiones regulares para definir patrones de tokens.
*Yacc: Generador de analizadores sintácticos ascendentes que construye el árbol de análisis implícitamente a través de operaciones shift-reduce.
*Diferencia: El ASDR del compilador Micro fue construido manualmente, mientras que Lex y Yacc automatizan el proceso de creación de los analizadores, lo que facilita el desarrollo de compiladores más complejos.