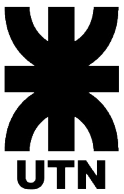
Universidad Tecnológica Nacional



Facultad Regional Buenos Aires

Sintaxis y Semántica de los Lenguajes

Compilador de lenguaje Micro

Autores:
Franco Battaglia
Lautaro Martínez Larroulet
Kevin Vargas Navia
Guillermo Verdile

Profesor:
Dr. Oscar Bruno

23 de octubre de 2017

Índice

Prefacio

El código correspondiente a este trabajo, con documentación de apoyo, se encuentra almacenado en **https://github.com/francobatta/compiladorMicro**. Allí existen dos branches de código distintas: **master** almacena los analizadores léxico, sintáctico y semántico; **síntesis** agrega la parte correspondiente al código. La bibliografía consultada fue:

- *Muchnik, Jorge Daniel*. **Sintaxis y semántica de los lenguajes: desde el compilador.** - 1º ed. - Buenos Aires: Centro de Estudiantes de Ingeniería Tecnológica - CEIT, 2010.
- Documentación de Bison https://www.gnu.org/software/bison/manual/

Parte I
Especificación del lenguaje Micro

El lenguaje sobre el cual trabajamos se llama Micro y fue creado por Charles Fischer. Micro posee una estructura muy simple que permite realizar de forma simple la construcción de un compilador básico que lo represente. Informalmente, Fischer describe al lenguaje Micro como:

- 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.
  - 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.

1. Definición formal

No obstante, para realizar la construcción de un compilador se necesita una definición formal. La misma consiste, en el caso de Micro, de una gramática *léxica* (que define los posibles lexemas que se pueden encontrar en un programa Micro) y una grámatica *sintáctica* (que permite precisar las construcciones válidas en Micro).
Para la implementación del compilador es muy importante la colaboración de ambas especificaciones: la sintáctica recibirá los lexemas de la léxica, para aplicarlos a las reglas de construcción propias. En nuestro caso, la especificación formal sintáctica también se encarga de realizar comprobaciones semánticas, y por último, de realizar una síntesis del código fuente Micro a código C.

1.1. 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 **( ) , ;**
Los lexemas que constituyen un programa fuente en este LP Micro son: identificadores, constantes, las cuatro palabras reservadas ya mencionadas, paréntesis izquierdo y derecho, el "punto y coma" (para terminar cada sentencia), la "coma" (para formar una lista), el símbolo de asignación (:=), y los operadores de suma (+) y de resta (−).

1.2. Gramática sintáctica

<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>)

Tabla de Tokens (a modo informativo)

En el programa fuente	Nombre del Token
Inicio	INICIO
Fin	FIN
Leer	LEER
Escribir	ESCRIBIR
:=	ASIGNACION
(	PARENIZQUIERDO
)	PARENDERECHO
,	COMA
;	PUNTOYCOMA
+	SUMA
−	RESTA

Parte II
Analizador lexicográfico

Un analizador léxico, analizador lexicográfico o *Scanner* es la primera fase de un compilador consistente en un programa que produce una salida compuesta de *tokens* (componentes léxicos) o símbolos.

1. Marco teórico

El análisis léxico es realizado por un módulo llamado Scanner. Este analizador lee, uno a uno, los caracteres que forman un programa fuente, y produce una secuencia de representaciones de Tokens. Es muy importante tener en cuenta que el Scanner es una rutina que produce y retorna la representación del correspondiente token, uno por vez, en la medida que es invocada por el *Parser*. Existen dos formas principales de implementar un Scanner:

1. A través de la utilización de un programa auxiliar tipo lex, en el que los datos son Tokens representados mediante Expresiones Regulares (método usado en este trabajo),
2. Mediante la construcción de una rutina basada en el diseño de un apropiado AFD (Autómata Finito Determinístico).

2. Herramienta utilizada

Para realizar el analizador léxico se usó el programa flex[1], que representa una alternativa de código abierto derivada del software lex. Flex recibe un archivo de extensión `.l` que especifica, de forma abstracta, la gramática léxica deseada, y construye como salida un archivo llamado `lex.yy.c` que contiene la gramática en forma de código fuente C.
Para compilar un archivo de entrada usando flex, basta con escribir `flex «ruta entrada»` desde la línea de comando del sistema.

3. Estructura de la entrada

La estructura del archivo de extensión `.l` usado como entrada se divide en tres secciones separadas por el símbolo `%%`. Las secciones representan, en el orden correspondiente:

1. Inclusión de **macros** y **librerías** de C, declaración de variables.
   Definición de **tokens** en la forma «`nombre símbolos_asociados`» separados por un renglón.
2. Definición de **reglas:** meta expresiones regulares formadas por tokens y operadores asociados a las metaER, acompañadas por porciones de código C asociadas a la detección de dichas metaER. Son de la forma «`regla {código_C}`» separadas por un renglón.
3. Inclusión de **funciones** y **procedimientos** escritos en C, sobre todo útiles para ser llamados en la sección 2 para brindar información complementaria sobre la detección de tokens. Nota: una vez hecha la especificación sintáctica, esta sección deja de tener sentido y se desarrolla en dicha especificación.

[1] Fast lexical analyzer generator. Descarga para Windows en http://gnuwin32.sourceforge.net/packages/flex.htm

4. Aplicación a Micro

Teniendo en cuenta toda la información mencionada anteriormente (haciendo hincapié en la gramática léxica de la sección 1.1), se escribió esta especificación léxica:

```
/*El Option noyywrap Avisa que solo leera un archivo */
%option noyywrap
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sintactico.tab.h"
%} /* Finaliza inclusion de librerias*/
LETRA [a-zA-Z]
DIGITO [0-9]
OPERADOR_ADITIVO [+|-]
ASIGNACION :=
LETRA_DIGITO ({LETRA}|{DIGITO})
/*SEPARADOR sirve para ignorar espacios en blanco*/
SEPARADOR ([ \t\n])+
void yyerror(char *);
%% /* Finaliza declaracion de tokens*/
inicio {return INICIO;}
fin {return FIN;}
leer {return LEER;}
escribir {return ESCRIBIR;}
"(" {yylval.cadena= strdup(yytext);return PI;}/* yylval=strdup... sirve para que se
    pueda realizar el proceso de sintesis*/
")" {yylval.cadena= strdup(yytext);return PD;}/* yylval guarda el contenido de la
    deteccion para su futuro uso */
"," {return COMA;}
";" {return PUNTOCOMA;}
{DIGITO}+ {yylval.cadena= strdup(yytext);return CONSTANTE;}
{LETRA}{LETRA_DIGITO}{0,31} {yylval.cadena= strdup(yytext);return IDENTIFICADOR;} /*
    Limite de 32 caracteres para los ID */
{OPERADOR_ADITIVO} {yylval.cadena= strdup(yytext);return OP_ADITIVO;}
{ASIGNACION} {return ASIGNACION;}
{SEPARADOR} {} /* Ignora espacio en blanco */
. {yyerror("Caracter Desconocido, ");} /* El punto detecta cualquier input. Debido a la
     naturaleza secuencial del analizador, se pone a lo ultimo*/
%% /* Finaliza reglas.*/
/* Parte en C implementada en especificacion sintactica */
```

lexico.l

Esta especificación no contiene mayores complicaciones, sino que intenta asemejarse a la especificación abstracta lo más posible. En particular notar que las palabras reservadas se escriben como reglas directamente. La tercera parte del archivo está vacía debido a que se implementa directamente en la especificación sintáctica (que posee estructura similar). Debido a esto, el archivo `.l` se desprende de toda comprobación ajena a la léxica: los roles semánticos y de síntesis estarán ligados, veremos más adelante, al analizador sintáctico.

Parte III
Analizador sintáctico

Un analizador sintáctico (o Parser) es un programa informático que analiza una cadena de símbolos de acuerdo a las reglas de una gramática formal. El parser procesa los tokens provistos por el analizador léxico para construir la estructura de datos.
Nota importante: en esta documentación elegimos separar las explicaciones del analizador sintáctico de los procesos semántico y de síntesis (si bien el código no realiza estas distinciones), para facilitar la comprensión y separación de la gramática sintáctica respecto de las funciones de C que les producen efecto.

1. Marco teórico

El análisis sintáctico es realizado por un módulo llamado Parser. Este analizador procesa los tokens que le entrega el Scanner hasta que reconoce una construcción sintáctica que requiere un procesamiento semántico.
Entonces, invoca directamente a la rutina semántica que corresponde. Algunas de estas rutinas semánticas utilizan, en sus procesamientos, la información de la representación de un token, como veremos más adelante. Nótese que no existe un módulo independiente llamado Analizador Semántico, sino que el análisis semántico se va realizando, en la medida que el Parser lo requiere, a través de las rutinas semánticas.
Las rutinas semánticas realizan el Análisis Semántico y también producen una salida en un lenguaje; esto último forma parte de la etapa de Síntesis del compilador.

2. Herramienta utilizada

Para realizar el analizador sintáctico (y luego veremos semántico y de síntesis) se usó la herramienta bison[2] que convierte la descripción formal de un lenguaje, escrita como una gramática independiente de contexto GIC, en un programa en C que realiza análisis sintáctico. Esta especificación se recibe en forma de archivo con extensión `.y` y construye como salida un archivo de extensión `.tab.c`, que al ser compilado en C constituye el analizador sintáctico correspondiente.
Para compilar un archivo de entrada usando bison, basta con escribir `bison -d «ruta entrada»` desde la línea de comando del sistema.

3. Estructura de la entrada

La estructura del archivo de extensión `.y` (similar a aquella del archivo `.l`) usado como entrada se divide en tres secciones separadas por el símbolo `%%`. Las secciones representan, en el orden correspondiente:

1. Inclusión de **macros** y **librerías** de C, declaración de variables.
   Declaración de los tokens (especificados en léxico) en la forma `%token «nombre token»` separados por un renglón.

2. Definición de **reglas:** gramática formada por composición de tokens, acompañadas por porciones de código C asociadas a la detección de dichas estructuras. Son de la forma `«gramática {código_C}»` separadas por un renglón. Con fines didácticos, en esta parte del trabajo no se muestran dichas reglas.

[2]Descarga para Windows en http://gnuwin32.sourceforge.net/packages/bison.htm

3. Inclusión de **funciones** y **procedimientos** escritos en C, que proporcionan un menú al analizador sintáctico, y le proveen funcionalidades adicionales como detección de archivos por línea de comando o reporte de errores detallado. Más adelante se incluirán funciones de carácter semántico y de síntesis en esta sección.

4. Aplicación a Micro

Teniendo en cuenta la información de la Parte I (sobre todo la gramática sintáctica provista), se escribió este analizador sintáctico:

```
%{
/* Analizador Sintactico */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
extern int yylex(void);
extern char *yytext;
extern FILE *yyin;
void yyerror(char *s);
void yyerrors(int tipo,char *s);
%} // Finaliza inclusion de librerias
%error-verbose // Reporte detallado de errores
%start Programa
%token INICIO
%token FIN
%token IDENTIFICADOR
%token LEER
%token ESCRIBIR
%token ASIGNACION
%token PUNTOCOMA
%token CONSTANTE
%token COMA
%token PI
%token PD
%token OP_ADITIVO
%% // Finaliza listado de tokens lexicos
// Inicio de gramatica sintactica
Programa: INICIO ListaDeSentencias FIN {YYACCEPT;}
ListaDeSentencias: Sentencia
    |ListaDeSentencias Sentencia
Sentencia: IDENTIFICADOR ASIGNACION Expresion PUNTOCOMA
  |LEER PI ListaIdentificadores PD PUNTOCOMA
  |ESCRIBIR PI ListaExpresiones PD PUNTOCOMA
ListaIdentificadores: IDENTIFICADOR
  | ListaIdentificadores COMA IDENTIFICADOR
ListaExpresiones: Expresion
  |ListaExpresiones COMA Expresion
Expresion: Primaria
  |Expresion OP_ADITIVO Expresion
Primaria: IDENTIFICADOR
  | CONSTANTE
  | PI Expresion PD
// Fin de gramatica sintactica
%%
// Menu del analizador sintactico creado
void yyerror(char *s)
{
printf("Error %s",s);
```

```
}
int main(int argc,char **argv)
{
if (argc>3 || argc<1)
{printf("Error, cantidad incorrecta de parametros");
getch();
return 1;}
else if (argc==2) // Permite pasaje de archivo por linea de comando
 {yyin=fopen(argv[1],"rt");
 printf("Archivo %s abierto\n",argv[1]);}
else
 {printf("Por favor ingrese programa a ser analizado\n");
 yyin=stdin;} // Analiza input del usuario
switch(yyparse()){
case 0:
 printf("Compilado Correctamente\n Presione una tecla para salir..."); break;
case 1:
 puts("\nCompilacion abortada"); break;
case 2:
 puts("Memoria insuficiente"); break;
} // Manejo del comando yyparse segun documentacion de bison
getch();
return 0;
}
```

sintactico_noSemantico_noSintesis.y

Análogamente a la parte léxica, esta especificación sintáctica es muy parecida a la provista por la bibliografía y no plantea cambios radicales. La inclusión del comando `%error-verbose` da un interesante complemento al reporte de errores por parte del analizador sintáctico. El manejo de la función `yyparse()` surge de la documentación de bison, que incluiremos como bibliografía.

Parte IV
Analizador semántico y proceso de síntesis

El análisis semántico, del cual ya hemos hablado anteriormente, es la tercera fase de análisis que existe en un compilador. Por un lado, el análisis Semántico está inmerso dentro del análisis Sintáctico, y, por otro lado, es el comienzo de la etapa de Síntesis del compilador.
La etapa de síntesis constituye la traducción del archivo fuente ingresado correctamente en una especificación de bajo nivel para una máquina virtual. En este trabajo elegimos realizar una síntesis (traducción) a lenguaje C, dado que es muy fácil compilarlo posteriormente.

1. Consideraciones

Debido a que Micro es un lenguaje de carácter didáctico, no posee muchas restricciones semánticas: por ejemplo, las variables son declaradas implícitamente. Es por ello que nos fue posible generar, con una cantidad relativamente pequeña de funciones C, comprobaciones semánticas importantes para el funcionamiento del compilador.

2. Implementación en Micro

```
%{
/* Analizador Sintactico */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
extern int yylex(void);
extern char *yytext;
extern FILE *yyin;
void yyerror(char *s);
typedef struct Nodo{
    char cadena[33];
    struct Nodo* siguiente;
} Nodo;
typedef struct Lista{
    Nodo* cabeza;
} Lista;
Lista* inicializacion();
Nodo* crearNodo(char *a);
void insertarIdentificador(Lista *lista,char *a);
void yyerrors(char *s);
void inicializarArchivo(FILE *f);
int estaEnLista(Lista *lista,char *a);
void agregarIdentificadorArchivo(FILE *f,char *s);
void asignarValorIdentiAarchivo(FILE *f,char *s,char *a);
void agregarLeerArchivo(FILE *f,char *a);
void agregarEscrbirArchivo(FILE *f,char *a);
void finalizarArchivo(FILE *f);
Lista *lista;
FILE *f;
%} // Fin prototipo funciones
%union {
  char * cadena;
} // Declara tipo de dato char como <cadena>
```

```
%error-verbose
%start Programa
%token INICIO
%token FIN
%token <cadena>IDENTIFICADOR
%token LEER
%token ESCRIBIR
%token ASIGNACION
%token PUNTOCOMA
%token <cadena>CONSTANTE
%token COMA
%token <cadena>PI
%token <cadena>PD
%token <cadena>OP_ADITIVO
%type <cadena>Expresion
%type <cadena>Primaria
%% // Fin declaracion tokens
Programa: INICIO ListaDeSentencias FIN {YYACCEPT;}
ListaDeSentencias: Sentencia
    |ListaDeSentencias Sentencia
Sentencia: IDENTIFICADOR ASIGNACION Expresion PUNTOCOMA {if(estaEnLista(lista,$1)){
    asignarValorIdentiAarchivo(f,$1,$3);}
                            else{insertarIdentificador(lista,$1);
    agregarIdentificadorArchivo(f,$1);
                              asignarValorIdentiAarchivo(f,$1,$3);}}
                            // Rutina de asignacion
  |LEER PI ListaIdentificadores PD PUNTOCOMA
  |ESCRIBIR PI ListaExpresiones PD PUNTOCOMA
ListaIdentificadores: IDENTIFICADOR {if(!estaEnLista(lista,$1)){insertarIdentificador(
    lista,$1);agregarIdentificadorArchivo(f,$1);}
                  agregarLeerArchivo(f,$1);}
                            // Rutina de lectura
  | ListaIdentificadores COMA IDENTIFICADOR {if(!estaEnLista(lista,$3)){
    insertarIdentificador(lista,$3);agregarIdentificadorArchivo(f,$3);}
                        agregarLeerArchivo(f,$3);}
ListaExpresiones: Expresion {agregarEscrbirArchivo(f,$1);}
  |ListaExpresiones COMA Expresion {agregarEscrbirArchivo(f,$3);}
                            // Rutina de escritura
Expresion: Primaria
  |Expresion OP_ADITIVO Expresion {$$=strdup(strcat(strcat($1,$2),$3))}
Primaria: IDENTIFICADOR {if(!estaEnLista(lista,$1)){yyerrors($1);YYABORT;}}
  | CONSTANTE
  | PI Expresion PD {$$=strdup(strcat(strcat($1,$2),$3))}
%%
void yyerror(char *s)
{
printf("Error %s",s);
}
// Inicio funciones lista
Lista* inicializacion(){
  Lista *lista=(Lista *)malloc(sizeof(Lista));
  lista->cabeza = NULL;
  return lista;
}
Nodo* crearNodo(char *a){
    Nodo* nodo = (Nodo *)malloc(sizeof(Nodo));
    strncpy(nodo->cadena,a,sizeof(nodo->cadena));
    nodo->siguiente=NULL;
    return nodo;
}
// Fin funciones lista
// Inicio funciones semanticas
```

```
93 void insertarIdentificador(Lista *lista,char *a){
       Nodo* nodo=crearNodo(a);
95 if(lista->cabeza== NULL){
       lista->cabeza=nodo;
97 }
   else{
99      Nodo* aux=lista->cabeza;
        while(aux->siguiente){
101             aux = aux->siguiente;
        }
103     aux->siguiente=nodo;
   }
105 }
   int estaEnLista(Lista *lista,char *a){
107   Nodo *aux=lista->cabeza;
      if(aux==NULL){
109     return 0;
      }
111   else{
        while(aux!=NULL){
113       if(strcmp(aux->cadena,a)==0)
          return 1;
115       else
          aux=aux->siguiente;
117     }
        return 0;
119   }
   }
121 void yyerrors(char *s){
       printf("Identificador %s no declarado ",s);
123 }
   // Fin funciones semanticas
125 int main(int argc,char **argv)
   {
127 int ret;
   lista=inicializacion();
129 inicializarArchivo(f);
   if (argc>3 || argc<1)
131 {printf("Error, cantidad incorrecta de parametros");
   getch();
133 return 1;}
   else if (argc==2)
135  {yyin=fopen(argv[1],"rt");
     printf("Archivo %s abierto\n",argv[1]);}
137 else
     {printf("Por favor ingrese programa a ser compilado\n");
139  yyin=stdin;}
   switch(yyparse()){
141 case 0:
     printf("Compilado Correctamente\n Presione una tecla para salir...");finalizarArchivo(
       f); break;
143                 // Rutina Fin
   case 1:
145   ret = remove("salidaEnC.c"); // Borrar archivo salida ante error
     puts("\nCompilacion abortada"); break;
147 case 2:
     puts("Memoria insuficiente"); break;
149 }
   getch();
151 return 0;
   }
153 // Inicio funciones de sintesis
```

```
void inicializarArchivo(FILE *f){
155   f=fopen("salidaEnC.c","w");
   fprintf(f,"/*Sintesis del programa en micro en C*/\n");
157   fprintf(f,"#include <stdio.h>\n");
   fprintf(f,"#include <stdlib.h>\n");
159   fprintf(f,"int main(){\n");
   fclose(f);
161 }
void agregarIdentificadorArchivo(FILE *f,char *s){
163   f=fopen("salidaEnC.c","a+");
   fprintf(f,"int %s;\n",s);
165   fclose(f);
}
167 void asignarValorIdentiAarchivo(FILE *f,char *s,char *a){
   f=fopen("salidaEnC.c","a+");
169   fprintf(f,"%s = %s;\n",s,a);
   fclose(f);
171 }
void agregarLeerArchivo(FILE *f,char *a){
173   f=fopen("salidaEnC.c","a+");
   fprintf(f,"scanf(\"%%d\",&%s);\n",a);
175   fclose(f);
}
177 void agregarEscrbirArchivo(FILE *f,char *a){
   f=fopen("salidaEnC.c","a+");
179   fprintf(f,"printf(\"%%d\\n\",%s);\n",a);
   fclose(f);
181 }
void finalizarArchivo(FILE *f){
183   f=fopen("salidaEnC.c","a+");
   fprintf(f,"\nreturn 0;\n}");
185   fclose(f);
}
187 // Fin funciones de sintesis
```

sintactico.y

Reglas Agregamos piezas de código C para realizar distintas acciones. El bison usa una notación posicional de izquierda a derecha para determinar el token sobre el cual se opera: `$$` quiere decir "resultado", y `$N` indica según el número $N = 1, 2, 3 \ldots$ la posición del token deseado. Por ejemplo, en la línea 55 se llama al token `$1` para comprobar si se encuentra en la lista, entendiendo que `$1` en aquel caso es un identificador.

Listas La colección de identificadores resulta ser una lista implementada en C: para ello, desde la línea 79 hasta la 89 se pueden ver las funciones de creación de lista.

Semántica Elegimos armar listas con funciones de C para poder almacenar identificadores y poder compararlos entre ellos; así, por ejemplo, es posible determinar si una rutina `escribir(...)` intenta escribir un identificador no inicializado anteriormente. Otra comprobación interesante es evitar la doble asignación `:=` de una variable, o chequear si una rutina `leer(...)` declara de forma implícita o si recibe una variable inicializada anteriormente. Desde la línea 92 hasta la 119 se encuentran las funciones que realizan comprobaciones semánticas con ayuda de las reglas.

Síntesis Las funciones de síntesis se encuentran desde la línea 158 hasta la última línea del programa. Las mismas se encargan de administrar un archivo de texto de salida, al cual se le va agregando la información

pasada por parámetros de funciones. El llamado a estas funciones se realiza dentro de cada regla (por ejemplo, en la línea 65 se llama a la función `agregarEscribirArchivo(f,...)`) y de forma secuencial, es decir, luego de cada token se ejecuta la síntesis correspondiente. Una ventaja de esta implementación es que la síntesis desconoce los pormenores del análisis sintáctico y semántico, ya que la misma es llamada dentro de estructuras de control de las reglas.

3. Conclusión

La construcción de este compilador fue un proceso gradual, en el cual nos fue muy práctico desarrollar en primer lugar el analizador sintáctico. Debido a la naturaleza de Micro, este análisis impone restricciones fuertes para los futuros errores semánticos. El análisis semántico fue sencillo de aplicar, pero resultó necesario crear estructuras más complejas (implementación en listas). Por último, la síntesis fue un proceso independiente del análisis y un agregado de interés para crear una salida funcional en C.